New research activities sailing under the brands of semantic web, semantic web service, and semantic computing have extended, and partly also confused the classical meaning of the term semantics as the software engineering community established it in the last century. In this article we try to shed some light on the different connotations of meaning with this word. We reflect on the role of semantic definitions and formally defined specifications, modeling and programming languages in software engineering activities. We sketch formally defined construction and validation methods, and discuss contributions of tools that exploit semantic information to enhance the quality of software products and development processes. We explore recent work on the use of semantic computing technology in software engineering and discuss opportunities for successful future applications. We conclude with an outlook on the potential of service-oriented computing to change the way software applications are designed, laid out, delivered, and used.

Keywords: semantics in software engineering; formal methods; semantic computing in software engineering.

1. Introduction

A contemporary interpretation of the concept “semantic software engineering” suggests the application and evolution of semantic computing technologies to all kinds of software artifacts being produced and consumed in a software development process. Semantic computing is a relatively new buzzword denoting an area of research which attempts to unify a range of computational techniques that allow (a) the extraction of information from various types of multimedia data, including text,
speech, image, or video, and (b) the meaningful correlation of information fragments originating from disparate sources.

Semantic computing has a broader scope than semantic web research, both in breadth and depth. The semantic web focuses on the augmentation of web documents through machine-readable metadata and related processing methods. The semantic web promotes a formal data representation, thus allowing software agents to grasp some of the intended meaning of data on the web and meaningfully integrate data from disparate web resources. To achieve these objectives, semantic web technologies are based on model theory to provide semantics of knowledge representation techniques and logics to perform inferences on web data.

Semantic computing addresses the information content of all possible data sources occurring in IT applications, not just web pages. It builds on a range of base technologies including natural language understanding and processing, speech recognition, text and data mining techniques, or formal concept analysis.

1.1. Software Engineering

Software engineering is the methodology of software construction. It provides concepts, processes, models, languages, tools, and principles for building, operating, and evolving software exhibiting desired qualities such as dependability, usability, correctness, robustness, or ease of maintenance. Software engineering also provides methods and tools that allow developers to predict and prove expected qualities and behavior of a piece of software.

Software performs different functions for society. It can be an independent product like a database system, a tax program, or security software; it can be embedded in technical products ranging from industrial robots to mobile telephones. Software can be used to control complex production processes or traffic systems, or it can provide the substrate for a diversity of service businesses in tourism industry, banking, or health care. But software is not just an executable program. It is a solution for an application problem that needs to serve its intended purpose, satisfy the stakeholders’ requirements, and the constraints imposed by technical, economic, and social conditions.

Software engineering provides the means to elicit the requirements for any possible application, to systematically transform these requirements into executable programs, and to establish or reconstruct relationships between different artifacts produced throughout the development process. These artifacts encode problems-solving knowledge in many different forms and at different levels of abstraction: requirements specifications, design and architecture models, software component interfaces, data structures, programs and all kind of informal documents including e-mails, meeting minutes, interview notes, memos, diagrams, or dictionaries.

*Individuals, groups of persons, or organizations who have a vested interest in the planned software in their role as contractors, users, or experts in related areas like sales, legal regulations, or accounting.
1.2. Semantics and Formal Methods

Viewed from a traditional software engineering perspective, “semantic software engineering” connotes mathematical approaches to software development based on specification, modeling and programming notations, and languages with mathematical semantics and formally defined development methods such as refinement [94], reification [22, 28], or transformational programming [6]. The formal semantics of a language require three parts [44]: a formally defined syntax, a semantic domain that provides a mathematically defined model suitable to represent all concepts of the language to be defined, and a semantic mapping that relates the syntactic expression to the elements of the semantic domain. A huge body of literature has been subsumed under the heading formal methods, beginning in the late 1960s.

Some people even confuse XML, the extensible mark-up language [13], with semantics. But XML is just a language for specifying the syntax and (hierarchical) structure of other languages used to communicate, store, and transform data, i.e., XML is just a tool to deal with syntax.

The notion semantics exhibits several aspects that are all needed and used cooperatively in a semantic computing setting. The two central aspects are:

- **domain semantics**, describing the concepts occurring in a domain of discourse and their relationships, and
- **behavioral semantics**, describing formally what happens with those concepts when being manipulated or transformed by a program.

Since for any semantics we need a syntactic reification, we find widely used reifications of the domain semantics, e.g., in terms of class diagrams, information models, or — more recently — ontologies, all of them expressed in some syntax of choice. Examples of successful notations include the diagram types standardized in the Unified Modeling Language (UML) [78], the abstract and conceptual representation of data with entity-relationship diagrams [18], and the OWL Web Ontology Language [27], respectively. With these means we can state what “things” we are talking about, and what their inherent properties and relations are. Computation is also present here, e.g., via the description logics underlying OWL or relational algebra underlying queries in the relational databases that embody such information systems.

Behavioral semantics were traditionally coupled to programming languages. The choice of a language implied the determination of a behavioral semantics framework, and the compiler realized this framework implicitly: a translator between what you see (the program source code) and what you get (the execution on the machine). The choice could be coarse-grained, e.g., between different programming paradigms (working on objects or high-level functions), within a paradigm, or between different programming languages (e.g., inheritance schemes, static or dynamic binding, or more or less elaborate versions of type concepts). The choice could also be as fine-granular as the myriad of concurrency models that underlie concurrent pro-
gramming languages and that are often based on process calculi. Process calculi span from the Calculus of Communicating Systems (CCS) [72], which offers pairwise synchronization but lacks a possibility to reconfigure concurrent computations dynamically, to the $\pi$-calculus [73], which allows mutable communication links, probabilities, and other features. Process calculi distinguish a variety of notions of behavioral equivalence known as bisimulation, which intuitively mean that two models need to match each other’s transitions to be bisimilar. These behavioral semantics are often seen as synonym of a computation model. Accordingly, programming languages are provided (either from the beginning or retrofitted) with a formal description of their computations expressed in mathematical terms. Ideally, the compiler implements exactly this definition.

The semantics of a program are, then, given by what the instructions perform on the elements of the domain semantics they operate on. Both kinds of semantics are needed to achieve semantic software engineering.

1.3. Opportunities and Challenges for Semantic Computing in Software Engineering

As argued before, software engineering involves deep problem solving knowledge. Unfortunately, a great deal of problem solving knowledge acquired from software projects only exists in the heads of people, in the form of personal (e-mail) communication, internal memos, or otherwise vaguely formulated ideas and goals. Thus, a lot of important information is fleeting or forms an inaccessible part of the archive of project documents. Some software organizations have, therefore, introduced knowledge management in software engineering projects [90]. The core idea is to establish practices and tools to collect and codify information maintained in terms of denotations of concepts, of relationships between such concepts, and possibly context conditions. Opportunities for useful applications of semantic computing in this context include, e.g., content extraction from personal documents related to a software project and automatic computation and synthesis of domain ontologies from the collection of software artifacts.

An inherent problem of software engineering is the semantic gap between the stakeholders’ intentions about a software system originating and evolving in their heads and the ultimate programs, which attempt to address these intentions. Each program written in some language $L$ is a formal object because it comes with an unambiguous semantics: the semantic domain is the program code understood by a real or virtual machine on which the program is executed and the semantic mapping is established by the $L$ compiler (the fact that $L$ compilers from different vendors or for different platforms may exhibit slight variations in the semantic mapping is irrelevant for this argument). Intensions need to be made explicit to become the subject of communication and agreement between stakeholders and to be accessible for symbolic manipulation by computers. Besides interviews with stakeholders, brainstorm sessions, workshops, questionnaires, or use cases, existing
documents such as legal regulations, corporate policies or standards, and existing software are typical sources for requirements elicitation. Possible application areas for semantic computing technologies include:

- natural language interfaces, voice question and answer, and dialog management to capture requirements,
- observation and analysis of usage profiles and system behavior,
- mining of specifications, design and program documentation, and error reports,
- detection of semantic similarities between narrative requirement statements, and
- validation of requirement artifacts.

A third problem domain in which semantic computing technology may contribute is software reengineering. It aims at the reconstruction of an operational software system in two phases: reverse engineering, in which the system is analyzed to identify its components and their relationships and in which an abstract representation of the system is created [19], and a subsequent forward engineering phase. Section 6 will sketch a few successful applications of semantic computing to this field.

Software process engineering offers another field of research opportunities for semantic computing. Software process modeling arose in the late 1980s as a sub-discipline of software engineering [66] aiming at an explicit description of software development processes to enable effective reflection about a planned, on-going, or concluded process, to facilitate its reuse and evolution, and to support process management. At first, researchers focused on modeling core features of software processes like activities, tools, artifacts, and human or software agents using rule-based formalisms, Petri nets, programming languages, and many other techniques. Later experience showed that other issues like roles, organization of work, and people’s capabilities are of equal importance [1]. Semantic computing could be useful to address problems at the human-machine interface by deriving information content from information documents and communication processes.

### 1.4. Outline

The following sections are organized along the main phases and cross-sectional activities in software engineering. For each phase typical methods, activities, products, and challenges are discussed in search for opportunities to improve products and processes by applying formal methods and semantic computing techniques. Selected approaches will be presented to illustrate current achievements or promising lines of research.
2. Requirements Engineering

Requirements engineering is the first and one of the prime tasks in any software development project as the quality, adequacy, and stability of the artifacts it produces are decisive for the success of the whole project. It consists of three core activities: requirements elicitation, documentation, and agreement. Agreement refers to the act of consolidating requirements from different stakeholders, identifying and resolving conflicts between requirements.

2.1. Diagrammatic Notations for Requirements Capture

To overcome the deficits of informal requirement documents, such as ambiguity, omissions, or hidden inconsistencies and conflicts, Doug Ross proposed Structured Analysis and Design Technique (SADT)\(^b\) in the mid-1970s [89]. SADT comes with a graphic notation, a vocabulary for talking about the application, and a methodology for producing abstract descriptions of a software system. It is targeted at the specification of requirements for software systems. Its vocabulary includes concepts like activity, thing (object, data, document), interconnection (input, output, control, mechanism), and abstraction. SADT and related modeling notations had success as commercial requirements tools and, thus, evolved into the Integrated DEFinition (IDEF) standard\(^c\), which comprises a family of methods and support tools for function modeling, data modeling, capturing and documenting processes, object-oriented design, and ontology description.

Diagrammatic notations have been used since with UML being widely used in software practice today. There exists a rich body of literature about the formal semantics of (sublanguages of) UML, including the meta-model of UML published by the OMG, the standardization body. The metamodel is, however, not a semantic definition as discussed in Section 1.2 with the elements: syntax, semantic domain and semantic mapping; it just defines the abstract syntax of the UML notation (see also [44]). In contrast, the series of technical reports by Broy et al. entitled “Towards a System Model for UML” provide a semantics in the true sense [15].

2.2. Natural Language Requirements and Semantic Technologies

A recent study performed among German software companies confirmed the findings of earlier empirical analyses, namely, that textual documents drawn up in natural languages are still widely used for requirements documentation [111]. A number of projects accepted this observation as a challenge and proposed methods and tools to improve the quality of natural language documents. For example, Gnesi et al.

\(^b\)This is just a representative for a range of similar techniques like DeMarco's Structured Analysis (SA) or Gane’s and Sarson’s Data Flow Diagrams.

\(^c\)http://www.idef.com/. This and all web resources cited in this text were last retrieved 8 February 2009.
propose linguistic techniques able to detect ambiguities and contradictions in requirement statements and expose incompleteness [37]. Syntactic and morphologic analyses are used to detect vague or subjective and weak or implicit statements, respectively. The quality of the analysis thereby strongly depends on the precision and comprehensiveness of the domain dictionary referred to by the analysis algorithms.

In the 1990s, a number of projects developed requirements elicitation methods that put stakeholder goals in their focus. A range of taxonomies was developed that organized goals into classes. Some approaches associated utility and priority attributes with goals, others presented heuristic methods and related tools to refine goals applying problem reduction methods from AI, and some proposed to link non-orthogonal goals through “reinforces” or “weakens” relationships. KAOS (which stands for “Keep All Objectives Satisfied”), for example, is a comprehensive method to elicit requirements starting from goal modeling [26]\(^3\). Taxonomies of goals, constraints, objects, and actions are used in combination with rules and heuristic tactics to drive the elicitation process including the decision among choices. In [75] knowledge representation techniques from AI are employed and adapted in the presentation of a requirements modeling language that talks about entities, activities, and assertions. Assertions are expressed in a formal language. They can be used to specify properties of activities in terms of pre- and post-conditions and to associate invariants with entities.

The theme of 14th Monterey Workshop held in 2007, entitled “Innovations for Requirements Analysis: From Stakeholders’ Needs to Formal Designs”, exactly focused on the gap we addressed before: the difficulty to elicit formal requirements and design specifications from partly informal and semi-formal documents including informal meeting minutes, interview notes, memos, feature lists, use scenarios, UML diagrams, and more [111]. The workshop’s proceedings [81] include a number of articles that apply semantic computing techniques, e.g., to

- identify goals in stakeholder dialogs,
- process natural language documents in requirements engineering,
- evolve requirements models by combining ontologies and model weaving, and
- elicit requirements from blogs through text classification and machine learning.

Further opportunities for the use of semantic technology in requirements engineering include the association of domain ontologies with particular terms in various types of informal requirements documents. Ontologies represent both domain knowledge and metainformation to documents. This metainformation can be used by programs to make the search for particular statements more efficient, to correlate disparate documents — even if they use different notions to talk about the

\(^3\)This paper includes a broad discussion of contemporary related work.
same requirement — and to provide more sophisticated information content analyses and reasoning capabilities. Ontologies can add structure to a set of documents that seem unrelated from the perspective of a computer maintaining them. Ontology techniques can also be used to support traceability [88] across the artifacts produced throughout a software development and evolution process.

3. Modeling and Specification

Models play a central role in the engineering discipline as a tool for studying or analyzing properties, function, (mis-) behavior, architecture, feasibility, trade-offs, costs, and other important issues of systems, technical equipment, or appliances under survey. A model is a conceptual, physical, or mathematical abstraction portraying those elements, properties, and relations of a real system that are relevant for a particular viewpoint and that are part of the requirements specification. In software engineering, models can, e.g., represent the data structures, process, and laws valid for a particular application domain. An architectural model reflects the decomposition of a design into components and describes their roles, interfaces, and interactions. Models used for program analysis depict the internal structure of a program, data flow and control dependencies, traces and slices.

Modeling is the act of constructing a model for a specification of requirements. Modeling is a creative process that cannot be fully automated, in general. In software engineering practice, diagrammatic notations like entity relationship diagrams, UML, data flow diagrams, statecharts, and others are frequently used in modeling tasks. A drawback of many of these diagrammatic modeling notations is that they have no comprehensive mathematical basis and, thus, tool support for consistency and completeness checking and for reasoning and predicting certain properties or behavior of a system are largely lacking.

3.1. Formal Specification Notations and Methods

Computer science and mathematics provide a range of mathematical foundations including logic, (process) algebra, category theory, Petri net theory, model theory, and others on which modeling theories for discrete systems can be built. Models can better be expressed in terms of languages and notations that build on mathematical foundations. Specifications play a central role in the software development process [71, 103]. They are used for an exact, unambiguous, and complete recording of system requirements, they are subject of communication and contractual regulations between customers and developers, and they define requirements for later design, implementation and validation tasks and support systematic reusability of software components [112] as well as evolutionary extension and modification of software systems being in use [53].

The knowledge about the semantics of a formal specification language can be used for constructing semantic tools and tool generators [40]. Such tools are used for proving expected properties of a system specified, and for testing the adequacy
of a specification before beginning with further expensive development steps. To support inspection techniques, natural-language descriptions can automatically be generated from specifications. The first solution of this type was Swartout’s Gist English paraphraser [101]. Even in situations where formal specifications were only used for documentation purposes, their advantages over conventional activities have been proven by practitioners [46, 43].

Hundreds or thousands of formal specification notations have been invented during the last three decades. Many have disappeared from the scene, some have been successfully used in industrial applications, and some have even found their way into international standards. For instance, Milner’s CCS [72], a process algebraic calculus, and the algebraic specification technique ACT-One [30] were used in the definition of LOTOS [104]. LOTOS, the Language Of Temporal Ordering Specifications, is a formal description technique standardized by the International Standardization Organization (ISO).

As far too many choices exist today to cover them exhaustively here, we focus on a few prominent representatives of classes of formal specification notations and discuss core characteristics and limitations. Two 1996 state-of-the-art reports on formal specification languages by Clarke and Wing [20] and by Cooke et al. [23] provide further examples and further details on some of the formalisms we are going to touch below. An introductory text about formal methods can be found in [110]. Around the mid-1990s Hinchey and Bowen have also published a comprehensive anthology about applications of formal methods in [49]. A website entitled “Formal Methods Europe” is maintained at http://www.fmeurope.org/.

Algebraic techniques for the specification of abstract data types are the basis of an important and well known family of formal specification languages ([42, 32, 7]). They belong to the class of axiomatic methods because they define functional properties of software components by means of (conditional) equations or predicates, independently of the concrete representation of data and independently of the concrete implementation of the operations defined on data. To enhance their practical use, algebraic specification techniques support software engineering concepts, such as encapsulation, modularization [31], parameterization [38], error handling [39], and stepwise refinement [29].

Model-oriented approaches such as the Vienna Development Method (VDM) [9] or Z [96] belong to another class of formal methods. They differ from axiomatic methods by describing system behavior directly in the form of abstract models. These models are based on well known mathematical structures, such as sets, sequences, functions, or relations. VDM and Z seem to predominate in practical applications of formal methods, above all in the British software industry [46].

Algebraic specification languages, VDM, or Z are, however, not suited to describe the behavior of concurrent and distributed systems adequately. For this type of applications process-oriented methods, such as Petri nets [87], CCS [72], Hoare’s calculus of Communicating Sequential Processes (CSP) [50], temporal logic [63], or the parallel programming language Unity [17], provide proper modeling concepts.
such as process, action, interaction, or event. But for practical applications, these methods have the disadvantage of abstracting completely from data structures. Moreover, they often neglect modularization and stepwise refinement concepts.

To combine the strengths of data-oriented and process-oriented modeling concepts, a number of hybrid approaches has been proposed. The language LOTOS, for example, combines the algebraic specification of abstract data types with CCS-like and CSP-like process specifications [104]. The Raise system supports a specification language based on VDM and on CCS-like and CSP-like formulations [35]. The SEmi-GRaphical Specification language SEGRAS [57, 58] and Colored Petri Nets [51] integrate algebraic data specifications with Petri nets. A disadvantage of embedding abstract data types into LOTOS is that the proof methods developed for CSS are not applicable directly. Therefore, validation methods and tools for LOTOS are mostly restricted to the symbolic execution of process expressions, interpreting operations on data structures on the basis of the term substitution semantics of algebraic specifications. Similar arguments apply to SEGRAS and other hybrid languages. SEGRAS has also been used to define the semantics of a prototyping language for real-time applications [59] based on dataflows.

3.2. From Formal Specifications to Executable Code

Once a formal specification has been produced, several paradigms exist to evolve the specification into an executable software system. The left part of Fig. 1 depicts the options software developers found in the 1990s. Transformational programming or program synthesis denotes a mechanized software development process that starts with a formal specification and transforms it step by step into a procedural program using a set of formal transformation rules. These rules encode abstract programming knowledge such as algorithmic concepts, data type refinements, or program optimization rules. The correctness of the resulting program is guaranteed by construction as the semantics of the original specification is maintained in each transformation step. Three prominent examples dating back to the 1980s are CIP (Computer-aided, Intuition-guided Programming) developed by Bauer’s group at the Technical University of Munich, Kestrel Institute’s Specware system [54], and the ProCos project [65], which proposed refinement- and calculus-based transformations for real-time systems.

Automatic code generation realizes the vision to make programmers obsolete in the software development process. This vision dates back to the 1970s. It assumed that it would be possible to devise a specification language or user interface making it easy for people to express requirements precisely and completely, which could be transformed into code through an automated process. Around that time, several projects were started with the ambitious goal to implement automatic programming systems that would provide natural language interfaces, perform natural language dialogs with users, and mechanically derive executable programs from such dialogs using natural language understanding and processing techniques [47]. We
all know that this vision was overly ambitious and did not become true, as yet. In a paper entitled “A 15 Year Perspective on Automatic Programming” Balzer presents a somewhat more realistic version of this vision that was inspired by compilation techniques and resembles the transformational programming approach [4]. He proposed to build an interactive translation facility to evolve a high-level formal specification, which results from a requirements elicitation process, iteratively into a lower-level (i.e., more algorithmic) specification that could then be compiled into code. A few successful applications of this approach have been demonstrated in narrow domains, but it did not scale in breadth. More modern versions of automatic programming techniques use genetic algorithms or generative programming techniques with generic classes, templates, and prototypes to produce code automatically. However, their contribution to software practice can be neglected.

The traditional form of manual program construction from a specification needs to establish the correctness of the code with respect to the specification through a range of validation techniques including testing, and verification using mathematical proof techniques (see also Section 4). Verification obligations are particularly acute for safety- or security-sensitive applications [60, 76].

The graphic on the right side of Fig. 1 depicts a more current situation in which specification and model form the reference line for deriving large-scale software through a model-driven architecture (MDA*), which involves the use of generators to partly automate software development, or a service-oriented computing approach, which aims to build applications from services discovered on the web or in intranet registries.

4. Validation

Program validation underwent a significant evolution and diversification over the past 30 years. Most radical was the goal to establish correctness directly by construction. The “Mathematics of Program Construction” community developed for-
malisms and calculi that guarantee correctness-preserving transformation processes from high-level specifications to code (see previous section). Unfortunately, ambitious goals like establishing correctness by construction requires great skills and presents great variance from case to case, so that most of the proofs have to be carried out by expert hand. This approach is still adopted in some cases, but it does not scale, mainly due to a lack of comprehensive tool support and high intellectual requirements.

Verification-based approaches, on the contrary, adopt an a posteriori view: given a program, the task is to determine under which conditions it executes correctly. This is the line of thought behind popular methods like Hoare-style verification, Dijkstra’s weakest pre-condition method, and tools that embed and support them like the Karlsruhe Interactive Verifier (KIV).

A more abstract view to programs and their behavior was introduced by Cousot with his theory of abstract interpretations [25]. Instead of working directly on a full program, an abstract model of it is created. The model preserves or approximates in a controlled fashion the properties relevant for the analysis under consideration. Proofs can be carried out on the abstract model, which is typically much smaller and semantically more concise than the original program. In addition, the theory ensures that the proof results hold for the concrete program under consideration. The abstraction depends on the property under investigation. Therefore it provides an aspect-specific, global view on the program, amenable to simpler, typically tool-supported analyses. Combined with the progress in verification tools and environments, this opened the door for modern, large-scale software verification. Astrée [24], Cousot’s own tool suite for static analysis, has recently been in the ramp-light, since it is being used to analyze software for the Airbus flagship, the A380 aircraft.

Depending on the kind of abstraction and on the concreteness of the models, entire lines of research thrived. Common to them is the semantic approach, which means that the aspects characterized by the abstraction are semantic aspects.

4.1. Model Checking

Two very successful lines of research that are based on the abstraction principle are type checking and verification by model checking. Type checking establishes standard abstract interpretations in terms of types denoting properties of programming entities (data, functions, etc.) and rules for their compatibility. Some untyped languages (LISP, Erlang, . . .) still exist\(^8\). Model checking, whose inventors Clarke, Emerson, and Sifakis were honored with the Turing Award, uses abstractions in form of properties of the model under consideration. In this case, semantic abstraction can happen both on the model side and also on the property side, which again represents a specific aspect of observation. This way, decidability and scalability can

\(^1\)http://www.informatik.uni-augsburg.de/lehrstuehle/swt/se/kiv/

\(^8\)Already 1996 we observed that “model checking is the type checking of tomorrow” [99]
be much more easily achieved. Scalable model checking tools and frameworks exist since the late 1980s. The Edinburgh Concurrency Workbench [21] was characterized from the beginning by its breadth: it supports a variety of different verification methods, including equivalence checking, pre-order checking, and model checking, for several different process semantics.

A common abstraction that lends itself to model checking is the Boolean abstraction of programs. Pioneered in the “dataflow analysis as model checking” approach of Steffen [97], who first observed that dataflow analyses could be formulated in terms of temporal-logic formulae and that these can be checked on abstract versions of programs, it has more recently gained momentum, e.g., through the work of Schmidt [92] and Podelski [85]. Since then a whole generation of model-checkers and environments specifically tailored to software analysis evolved. For example, Bogor [10] is a customizable and extensible model-checker for object-oriented software that emphasizes the notion of domain-specific model-checking. Bandera is a tool suite for detecting hard-to-find defects in concurrent Java software using model-checking techniques [5].

Domain-specific approaches have also touched hardware verification. As hardware circuits are actually designed by describing them as programs in domain-specific languages like the Very High-speed integrated circuit Hardware Description Language (VHDL), Verilog, or SystemC, hardware verification is amenable to model-checking. Since the times of the invention of binary decision diagrams [2, 16] as a means to compactly encode huge state spaces, symbolic model-checking [70] and its implementation, e.g., in the SMV tool, have changed the way hardware circuits are verified at design time. Today, hardware producers routinely use this technique to ensure first-time correctness of their circuits. The success of model checking in the hardware domain encouraged attempts to apply this verification technique to software, too, by reducing the software to be checked to a (somehow hardware-equivalent) model with the same kind of binary encoding [41]. The transfer of methods and approaches between the two domains is, however, not as easy as it seemed at first, and experiences show that the differences lie in inherent semantic issues.

All these verification methods and approaches consider, however, software as its code basis (or its possibly abstract model): also component-based analysis is still oriented at code bases.

A significant breach with this mental category happens in the field of service-oriented computing (SOC), which will be discussed further in Section 7. SOC coupled with service level agreements introduces a new paradigm that breaks the direct dependence on code. It introduces granularities that are largely independent of the size and organization of the actual implementations, allowing for the first time full agility in the language used for talking and reasoning about systems. This leap can have a disruptive effect on the otherwise quite evolutionary path followed by software design and its semantics-oriented descriptions.
4.2. Higher Order Logic (HOL) for Verification

HOL denotes a family of interactive theorem-proving systems whose members use similar variants of higher-order logics [3]. A prominent member of this family is the generic proving assistant Isabelle [84]. Its inference rules are represented as generalized Horn clauses and inference is based on resolution. HOL and Isabelle have been used extensively to

- provide semantics to programming and specification languages,
- prove the correctness of formal (program or data) transformations [56, 74],
- verify compiler correctness [36]
- prove critical properties of safety-related programs [106], or
- reason about semantic web ontologies that were transformed into Isabelle/HOL theories [102].

Figure 2 taken from [105] illustrates the typical structure and components of verification processes performed with HOL systems.

The advantages of HOL include that the means of abstraction and quantification over functions make this logic very expressive and, thus, well suited to the concise description of complex theories. Evidence supporting this fact is provided by the embedding of hardware description languages [12] and the verification of floating-point algorithms [45].

HOL is a widely studied and well understood logical system with a remarkably small number of axioms and inference rules. Its expressiveness makes it possible to use definitional extension as the principal method of theory development. Since this method is conservative, logical inconsistencies can be practically ruled out.
Automatic type inference systems for HOL make type annotations unnecessary to a great extent. This shortens formulae and proofs, because the information contained in the typing is automatically inferred and propagated.

In comparison to alternatives such as Zermelo-Frankel set theory, which gave name to the Z notation [96], a few disadvantages must be conceded. The type discipline of HOL leads to a certain loss of flexibility, cf., e.g., [64]. This statement remains true despite the expressiveness of polymorphism and symbol overloading available in systems such as Isabelle/HOL. In comparison with first- and second-order logic, the implementation of the HOL type system is technically more demanding. In particular, the existence of type and function variables complicates unification, the basic method of equation solving [77]. In addition, most research in automated theorem proving has been performed in the area of first-order theories.

5. Component-based Software Engineering

Component-based software engineering (CBSE) aims to develop software from prefabricated parts, to enable their reuse, and to ease maintenance and customization of systems built from such reusable parts. The hallmark of component-based software engineering is a clear separation of program code, interfaces, and compositional architecture. A software architecture describes a software system as a structure of processing components and interactions between them. Components are encapsulated and can be organized hierarchically. Each component offers its application environment operations through one or more interfaces without betraying details of their implementation. Interfaces are the means by which components interconnect. Interfaces also define the contractual use of a component for composition with other components in an architecture. Beugnard et al. distinguish four levels of contract aspects [8]: (a) syntax (operation signatures and data types), (b) behavior (pre-, post-conditions, and invariants), (c) synchronization or interaction protocol, and (d) quality of service. Explicit interfaces permit integrators to deal with function, data, interaction, synchronization, and quality attributes of components at an abstract level without the need to look into the internals of reused components.

Fig. 3. Conformance constraints for a component architecture and component substitution
Core topics of component-based software engineering include: the composition of components to consistent architectures, the substitution of a component $C$ by a component $C'$ under certain conformance constraints, and compositional reasoning about functional and non-functional properties of architectures. The first issue is illustrated in Fig. 3 on the left with $P, P_i$ representing provided and $R, R_j$ requires interfaces, and $R \subseteq P_i$ stating that the requires interface $R$ must conform to the provided interface $P_i$ to allow for a link between the two interfaces [93]. The second issues is visualized in Fig. 3 on the right. These issues rely on semantic information including behavior, synchronization protocols, and quality metrics, and on reasoning capabilities. For instance, if the semantics of an interface’s protocol part are modeled in terms of trace semantics, conformance means that the traces generated by $R$ must observe the protocol of $P_i$. In this context, semantic computing solutions can contribute to deduce abstract information about a component’s behavior from the component implementation. In [93] the notion of design by contract from pre-, post-condition and invariant assertions on objects is extended to dynamic protocol descriptions for components. Based on formal models like finite state machines and Petri nets, it is shown how timing and reliability properties of a component assembly can be computed compositionally.

The development of architecture description languages with formal semantics began in the 1980s producing architecture languages like Darwin and Wright. Darwin is a configuration language for distributed systems\(^1\), where a component is the smallest unit of distribution. It encapsulates resources and has an interface that shows how many other components it can interact with. In their simplest form, the encapsulated resources can be single sequential processes or abstract data types. A component can also contain multiple threads. The semantics of Darwin have been defined using the $\pi$-calculus as a semantic domain [67].

In the Wright architectural specification language [34], a component definition consists of a list of port specifications for the interaction points and of a behavior specification for each component. A connector specification talks about roles, behavior, and an interaction protocol. Behavior and protocol are formally specified using language constructs from CSP [50].

6. Reengineering and Evolution

Once software systems are deployed, their adequate maintenance and evolution over a lifetime that may span decades represents one of the major sources of cost and risk for companies and organizations. Typically, maintenance is handed over to a different company, losing any link to the designers and to design time documents (specifications, models, implementation): in consequence, the native semantics are lost! Therefore, a significant line of research within software maintenance is concerned with the recovery of these semantics a posteriori. In the ideal case, the

\(^1\)http://www-dse.doc.ic.ac.uk/Software/.
source code is accessible and can itself be analyzed and queried for reengineering
and adaptation purposes.

In [95], Snelting applies formal concept analysis (FCA) to software reengineering
problems including the detection or assessment of module structures, the analysis
of configuration spaces, and the restructuring of class hierarchies of object-oriented
software. FCA [33] relies on the theory of ordered sets and complete lattices, and
provides mathematical methods and software tools to organize objects with common
attributes in groups based on lattice decompositions. Viewing a module as a set
of procedures and a set of variables on which these procedures operate, module
candidates result from the identification of relations between procedures and global
variables in a program. The work on refactoring class hierarchies combines FCA with
dataflow analysis and type inference to transform a given hierarchy in a semantically
equivalent one. This has the advantage that it accommodates (dynamic) member
access patterns.

Fig. 4 depicts a concept lattice taken from [82] that illustrates the partitioning
of web services from an open travel e-marketplace according to a domain ontology
covering airline and hotel booking, financial, insurance, and car rental services.

Practical experience, however, showed that FCA, in general, leads to concept
lattices that are highly connected, thus almost useless for refactoring in practical
cases.

More recent approaches use model extraction techniques, either from code, if
available, or directly from systems treated as black boxes. Model extraction from
code consists of analyzing the source code of a program and generating a statement
level control flow model analogous to those used by compilers in dataflow analysis.
Depending on their use, models can be more or less abstract. We target here on
cases of model-based migration, where a code basis is ported to a different plat-
form, or model-based repurposing, where layers of software are left unchanged or migrated (e.g., drivers steering some physical equipment), while others are being changed, reengineered, or added. In an on-going project, in which we are repurposing a business-critical application with 100,000 lines of C++ code, we surprisingly found out that the standard code analysis performed by compiler front-ends is already too abstract, and that it does not sufficiently capture the domain and behavior semantics. Therefore, a much finer analysis had to be carried out in order to faithfully respect these semantics [107].

A possible technique for extracting behavioral models from black-box systems combines testing with automata learning. LearnLib [86], for example, provides an entire framework that contains learning algorithms for a variety of automata. While learning, a system’s behavior is explored through systematic tests, and the evidence gained is combined into a minimum model consistent with all observations. This technique is limited to behavioral models. One discovers only what the system does at its observable interfaces. Its internal structure or architecture is not accessible. The technique is, therefore, not apt for reverse engineering in a complete sense, but quite adequate to extract a so-called cognitive model of a system, useful for specification, monitoring, and regression testing.

The problem of architectural design recovery from legacy code using data mining techniques and a variant of the branch-and-bound search algorithm is addressed in [91]. An architectural description of a system’s assumed components and component interfaces serves as a query applied to a large database of information extracted from legacy code.

In [52] the application of the Ontological Adaptive Service-Sharing Integration System (OASIS) is suggested to enable heterogeneous reengineering tools to cooperate based on shared services and, thus, to assist software engineers in performing software analysis and program comprehension tasks. OASIS is a tool integration approach that uses service-oriented tool adapters and a domain ontology to enable tool interoperation based on individual services. The domain ontology spans a concept space that controls the activation of a tool service. Two services $s_1$ and $s_2$ can be sequenced if the concepts required by $s_2$ are supported by $s_1$.

This resembles the notion of coarse-grained behavioral composition driven by abstract domain and behavioral semantics already proposed for Internet-based tool interoperation in [98]. The International Journal on Software Tools for Technology Transfer (STTT) is associated with an Electronic Tool Integration platform (ETI [14]) that serves as an active repository through which users can experiment with individual tools and investigate their interplay by combining them into complex heterogeneous systems that exploit various methodologies and algorithms. These activities are supported by libraries of benchmarks for tool evaluation and libraries of mediation modules for resolving interfacing problems between different tools and formats.
7. Service-oriented Computing

Service-oriented computing (SOC) is a transdisciplinary research area that relies on a range of disciplines including business computing, information management, software engineering, semantic web, artificial intelligence, and distributed systems. SOC builds on the idea of reusable computational elements that are encapsulated as services. Services are autonomous, platform-independent computational elements that can be described, published, discovered, orchestrated, and programmed using standard protocols. SOC conforms to the principles of a service-oriented architecture (SOA). SOA is an architectural style that aims at loosely coupled (dynamically) reconfigurable networks of services. The services in a network can be orchestrated by a formally defined workflow organizing the invocation of services and supporting sessions and transactions. The International Conference on Service Oriented Computing (ICSOC) is the flagship conference of the service engineering community that publishes outstanding research results and practice experiences in an annual LNCS series (see, e.g., [61, 11]).

SOC has also the potential for shifting the focus of software semantics from the producer-centric view, which primarily targets designers and programmers, to the service users who are typically non-IT professionals. One example for this change, which builds on top of semantic computing, is discussed in [80]. Paar and Tichy address the problem of web service discovery by presenting so-called soft look-up and semantic programming techniques that support natural language input in the process of service discovery. Wang et al. suggest to build application ontologies from descriptions of semantic web services [108] and use the semantics covered by such ontologies to enhance the precision of semantic web service discovery processes [109].

The reason why we are convinced that this opportunity for change opened up by SOC will be taken is economical: the SOC movement is backed and pushed not only by IT companies, producers of hardware, software, and communication systems, but it is also pulled by industry and organizations at large who demand a higher transparency and governability of their own, business-critical processes that this equipment embodies and enforces. In this light, we can regard service orientation as an 80/20 approach to process management that follows the “easy for the many, difficult for a few” principle, driven by semantics. Central features comprise:

- radical virtualization of all technical details unimportant for the service level,
- loose coupling of services to enable a maximum of freedom of choice, orchestration, and evolution of services, and
- domain specificity, to be directly usable by the addressed user community.

This effort, complemented by consequent standardization, directly increases the end user’s power. In [48] the essence of the service orientation viewpoint is nicely
summarized as:

\[
\text{semantic integration} + \text{loose coupling} + \text{managed evolution}
\]

The questions “What is easy?” and “Who are the few?” directly lead to a complex division of labor scheme, which is also reflected in the many subdisciplines, such as SO architectures, SO computing, SO programming, and SO engineering [83], all contributing as communities of “few” to ease the work of communities of “many.” In this sense, services becomes a synonym for semantic-driven and semantic-enhanced. That industry is willing and — to some extent — already able to follow, is demonstrated, for example, by two contributions to the IEEE Computer Special Issue on service orientation [69]:

- [55] discusses the current status of service orientation from the point of view of enterprises. Goal-driven business process compositions based on the enterprise physics metaphor should be generated on the basis of solid foundations that formalize web service descriptions, business policies, and business goals.
- [100] addresses the same issue, but from the point of view of methodologies and platforms that support an end-to-end and round-trip engineering experience at the user level. The central idea here is the “one-thing” approach, which enforces semantic compatibility between the different layers of refinement. This is a necessary precondition for enabling a round-trip engineering from the top level down to the physical realization and back that does not suffer from conceptual gaps between the things dealt with at the different levels.

A further, pretty much industry-driven development evolves around the concept of a “Service Component Architecture” (SCA) [79]. SCA allows an organization to exhibit its software assets as service components from which heterogeneous (cross-organizational) applications and systems can be composed in the spirit of component-based software engineering [62], where semantic issues occur in the need to extract semantic information from a service component’s code to make it visible at the component’s interface and observe semantic information in component interfaces in conformance checks when “wiring” components to subsystems, when substituting a component, or when computing non-functional properties of a system from properties of its constituent components.

SOC and SOA are not free of controversy, though. In particular, there is a great difference between delivering repackaged monolithic systems and true, semantically enabled service-oriented computing. As Burton Group analyst Manes writes [68]: “Except in rare situations, SOA has failed to deliver its promised benefits.” Most projects have focused on portioning and wrapping, and neglected the deep, disruptive impact that “thinking in service” brings to organizations and enterprises. The incremental architecture-driven approach has proven to be a short-range change, while the benefit is reaped only with a far reaching change of mind, that puts ser-
vice(s) and their purpose into the center of attention and makes them the driver of anything else around and below.

8. Conclusion

In this paper we tried to enlighten and illustrate the meaning of the different uses of the word “semantics” by different communities, in particular the software engineering and the semantic computing communities. We looked back into the history and role of mathematical formalisms as a basis for language, method, and tool design. We also presented opportunities and initial attempts to bring semantic computing techniques, in particular natural language recognition and processing and knowledge representation and reasoning facilities, into software engineering. Our current conclusion is that except for very few initial attempts, semantic computing did not yet penetrate into mainstream software engineering.

As an attractive overlap of interest area we identified a more recent software engineering paradigm relying on open standards and using services as basic building blocks. Semantic web services, a subdomain of service-oriented computing (SOC), heavily build on emerging semantic technologies. In this context ultra-light-weight formal methods could emerge that customers can understand.

References

[13] T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, and F. Yergeau (Eds.): Exten-
A Hindsight on Formal Methods and Prospects of Semantic Computing in Software Engineering


A Hindsight on Formal Methods and Prospects of Semantic Computing in Software Engineering


[105] N. Völker: *Ein Rahmen zur Verifikation von SPS-Funktionsbausteinen in HOL*, Shaker Verlag 1998 (Dissertation at FernUniversität in Hagen, in German)


