





Automated Test Production - Complement to “Ad-hoc” Testing

José Gomes and Luiz Dias

EasyChair preprints are intended for rapid dissemination of research results and are integrated with the rest of EasyChair.

May 22, 2022

Automated Test Production Complement to “Ad-hoc” Testing

José Marcos Gomes , Luiz Alberto Vieira Dias ,

Abstract—A view on software testing, taken in a broad sense and considered a important activity is presented. We discuss the methods and techniques for applying tests and the reasons we recognize make it difficult for industry to adopt the advances observed in academia. We discuss some advances in the area and briefly point out the approach we intend to follow in the search for a solution.

Index Terms—Software Engineering; Software Testing; Automated Test Production; Generative Testing; Test Driven Development.

I. MOTIVATION

Accepting that tests are important, but are not always implemented or kept up to date during the lifetime of a program, we conclude that nothing has changed since the introduction of the Agile Manifesto earlier this century [1] which we reproduce below and from which we highlight the passage “Software that works rather than complete documentation”[1].

- “Individuals and interactions over processes and tools”
- “Working software over comprehensive documentation”
- “Customer collaboration over contract negotiation”
- “Responding to change over following a plan”

This view has come to become an important industry trend [2]¹, where face-to-face interactions are preferable to formal communication processes and working programs are preferable to comprehensive documentation, leaving the interpretation of the term “comprehensive” to each agile development team to decide [4]. In fact the agile method suggests that all documentation can be replaced by informal communication with an emphasis on tacit rather than explicit knowledge [5].

On the other hand, the adoption of continuous integration and continuous delivery processes and tools has been steadily and unequivocally growing in both industry [6], [7] and open source projects [8], which can to some extent be interpreted as a denial of one of the principles of the Agile Manifesto: “Individuals and interactions over processes and tools”, yet this does not come as a relief to the fact that many see benefits in building and maintaining formal models, but are not content

Gomes, J.M. and Dias, L.A.V. are with Instituto Tecnológico de Aeronáutica.

This work was partially funded by the STAMPS project, a partnership between the Instituto Tecnológico de Aeronáutica, the Fundação Casimiro Montenegro Filho and Ecosystema Negócios Digitais

Manuscript received MMMMM DD, 2021; revised MMMMM DD, 2021.

¹The 14^o annual report STATEofAGILE from 2020 points out that 95% of organizations practice agile software development methods. [3].

to build them as they believe they consume too much time and resources, even believing in the slim chances of success of projects that do not use some modeling [9].

The implications of this view for the construction and maintenance of programs and the use and application of development methods and tools are discussed.

II. TEST PRODUCTION METHODS

The present discussion is a contribution to the understanding of how software testing fits into the present realities perceived by both industry and academia, even if these realities, as we shall see, do not correspond and will not converge. The *TDD (Test Driven Development)* technique is widely cited and recommended by the signers of the Agile Manifesto [10], even though it is not part of the manifesto or its twelve principles [1], so we can conclude that the *IT (Information Technology)* industry at least recognizes the importance of testing programs. The academia, on the other hand, perceives program testing based on formal specifications as inevitable in pioneering studies since the 1970s [11], [12], the foundations for combining formal methods and program testing being established and accepted, and it is up to the community to put them into practice, optimize and extend them.

In general, we classify the tests in **Formal**: verifiable by theoretical means or pure logic; and **Empirical**: verifiable through observation or direct experience².

A. Formal Testing

Hoare and Floyd introduced formal methods by introducing the “Hoare calculus” for proving the correctness of a program as well as the notions of pre and postconditions, invariants and assertions. His ideas were gradually developed into the current formal software engineering tools and techniques, such as the *OCL (Object Constraint Language)* [15] used to specify constraints in *UML (Unified Modelling Language)* diagrams.

According to Gaudel, for each and every specification method, there is a notation. Depending on the method, specifications can include expressions in various logical forms, used to write pre and postconditions, axioms of data types, constraints, temporal properties. They can represent definitions of process states, such as:

- *CSP (Communicating Sequential Processes)* [17]
- *CCS (Calculus of Communicating Systems)* [18]

²We take into account the formality of the test and not the conduct of the test, as it is perfectly possible to conduct empirical tests by adopting formal practices in their execution.

- *LOTOS (Language Of Temporal Ordering Specification)* [19]
- *Circus* [20]

Or they can have annotated diagrams, such as:

- *FSM (Finite State Machine)* [21]
- *LTS (Labelled Transition Systems)* [22]
- Petri Networks [23]
- etc.

But there is more than a syntax. First, there is a formal semantics, in terms of mathematical notions such as:

- Predicate transformers for pre and post conditions
- Classified sets and algebras for axiomatic definitions
- Various types of automata, traces, faults, divergences, for process algebras

Second, there is a formal deduction system, making it possible to perform proofs, or other checks (such as model checking), or both. Thus, formal specifications can be analyzed to guide the identification of appropriate test cases.

In addition to syntax, semantics, and the deduction system, formal methods come with some relations between specifications that formalize equivalence or correct step-by-step development. Depending on the context, such relations are called: refinement, conformance, or, in the case of formulas, satisfiability, and are fundamental to test methods [16].

Gaudel concludes that model-based tests are tests of the *black-box*³ type, where the internal organization of the program under test is ignored and the strategy is based on a description of the desired properties and behavior of the program⁴, which may be formal or not, or in other words, these methods target certain classes of faults and assume that the program is exempt from other types and classes of faults [16].

B. Empirical Tests

Without formal defined specifications *a priori*, which as we have seen in “Motivation” is a trend in the industry, we are left only with informal and empirical practice⁵ for the verification and validation of the correctness of computer program implementation⁶. One of the practices advocated by supporters of agile methods is *TDD*, where tests are written even before the program itself, but it does not show clear benefits⁷ compared to the option of implementing the tests after the program is ready [25]–[27], or it may be linked to the fact that processes like *TDD* encourage stable and refined steps of continuous improvement [28].

In the informal test, we have a relation of the hypothesis to an observation statement, which is nothing more than a

³Method of validating functional and external aspects of a computer application.

⁴We separate these tests into a category - that of **Formal Tests** - that is, tests with a formal basis and that originate from models.

⁵Which generally means: verifiable by direct observation or experience rather than by theory or pure logic, even though it is possible to adopt formal practices during an empirical procedure.

⁶Nothing prevents that, even starting from a basis of formal specifications, empirical tests be adopted in the verification of the implementation.

⁷The practice of *TDD* is advocated mainly because the alternative is to have no tests at all after the program is ready[24].

proposition about the perceptible properties of some entity, set of entities, or system, followed by a rule transmission where, if the observation statement directly confirms the hypothesis, then indirectly it confirms any of its logical consequences [29].

We can state that formal tests are cases of inductive inference⁸, and that in empirical tests we have a direct confirmation of the hypothesis, but without the soundness and precision that formal methods⁹ guarantee [30] because of the *ad hoc* attitude with which the informality of design¹⁰ of empirical testing is practiced.

Just as using only **Formal Methods** we are unable to judge all the possibilities of flaws that a program may present[31], we can state that **Empirical Methods** are also so, and for the same reasons, with the aggravating factor of introducing a certain randomness¹¹ to the process.

C. Static and Dynamic Analysis

This is a case where the test can either be defined *a priori* (as in *TDD* or model-based) or *a posteriori* (as most informal tests are done), and which according to Gaudel, would be the answer to the lack of coverage of Formal Tests, but which as we will see below, also present problems of application in practice.

Static analysis was introduced in 1980 with the work “*Methods to ensure the standardization of FORTRAN software. [PFORT, DAVE, POLISH, and BRNANL, for analysis and editing of codes, in FORTRAN for PDP-10 and IBM 360 and 370]*” by Gaffney and Wooten [32]. The nature of verification performed by static parsers include [33], [34] (but not limited to only these) the following analyses:

- *Layout* and source code formatting
- Identifying language constructs known to be non-portable
- Identifying algorithm constructs known to be unsafe
- Use of variables or constants with suspect names and contents (for example: **PASSWORD = 'SECRET'**)
- Detection of faults not considered by compilers
- Control flow analysis (detection of *loops*)
- Detect data usage in variables before a value has been entered
- Detect value overloading in variables (assign a very large value to a variable that only supports small values - in some languages assign a **DOUBLE** value to a simple **INT** variable)
- Detect memory overflow (leak) or the non-validation of may memory overflow (assigning a very long constant to a variable that supports a small memory size)
- Detect leakage of *handles* (the reference to the control structure) of files and accesses to communication resources
- Check permission to perform certain operations
- Ensuring the termination of a processing (or ensuring indications that it will not terminate)

⁸We cannot call “formal tests” a case of “indirect confirmation”.

⁹Formal methods pursue qualitative and quantitative metrics of the soundness and precision of the method itself.

¹⁰And as we said earlier, not necessarily of the actual conduct, which can be perfectly formal.

¹¹The observer’s objectivity and his judgment.

Classe	Descrição
Lexical Analysis	<i>Lexical analysis is based on the grammatical structure of the language. It divides the program into small parts that are compared to known fault libraries. Disregarding syntax, semantics and interaction between subroutines, the incidence of false positives is high [42].</i>
Type Inference	<i>It infers the type of variables and functions by the compiler or interpreter, and checks that accesses to these variables and functions conform to predefined rules for the type [43].</i>
Data Flow Analysis	<i>Refers to collecting semantic information from source code, and using algebraic method determines the definition and use of variables at compile time. Starting from the execution flow graph, a data flow analysis determines whether values in a program are flagged as potentially vulnerable variables [44].</i>
Rule Checking	<i>Checks the security of a program using pre-set rules [45]. Some rules, such as requiring execution under elevated privilege, carry security implications [42] and are detected.</i>
Constraints Analysis	<i>Divided between constraint generation and constraint resolution during the analysis process. Constraint generation sets variable types or analyzes the constraint system between different states of execution using predetermined rules; constraint resolution applies and resolves the generated constraints [42].</i>
Comparison of Correction Snippets	<i>Comparison of source or binary code snippets changed during the process of fixing flaws is used to find known implementation gaps. After patches have been applied to a program, the comparison serves to determine the location and causes of the vulnerability to which they apply [42].</i>
Symbolic Execution	<i>It represents program inputs as symbols instead of the actual data, and produces algebraic expressions over the symbol in the implementation process. By the constraint solving method symbolic execution can detect possible failures [46]–[49].</i>
Abstract Interpretation	<i>It is a formal description of program analysis, which maps the program to abstract domains. The technique requires completeness, which makes it impractical for very large programs, but proves correct for all possible inputs [50], [51].</i>
Proof of Theorems	<i>Semantic analysis of the program, which can solve infinite state system problems [52], [53]. First convert the program into a logical formula, then prove that the program is a valid theorem using axioms and rules [42].</i>
Model Verification	<i>Starting from formal models, such as state machines or directed graphs, it runs through them and compares the model with the implementation to see if it matches the characteristics predefined by the first [54].</i>

TABLE I: Classification of Static Analyzers

- Ensure the order in which processing is performed and terminated in a way that maintains the integrity of the information (or ensure that it gives indications that the information is not intact)
- Ensure that the process can be observed as deterministic¹² (or ensure that there are indications that the process cannot be observed as deterministic)

Many of these validations can be (and most often are) done by compilers (when the language is compiled)[35]. Since the purpose of the compiler is to generate executable code and not to check for programming faults, and other classes of faults can only be determined at runtime, such as memory overflow, which only occurs if a very long constant is supplied during program use,¹³ then specialized checkers such as **Linters**[36] are adopted. Capable of detecting a wide range of faults, including style (*layout* of source code), some use source code annotations to achieve better problem detection, at the expense of extra developer work [37]–[41].

Static analyzers can then be classified (see Table I) into various types and capabilities, covering the detection of several possible fault categories, from implementation to vulnerability and security related.

The problem with static analyzers is the high false positive rate (alerts that are not real problems), low understandability of alerts and lack of automation in quick fixes for the large

number of identified problems [55], such as: [56] code structure and [57] coding patterns, which could easily be fixed using automatic refactoring techniques [58], but as we will see below, the available tools are not in line with the latest advances made by the scientific community.

Dynamic analysis, on the other hand, is in contrast to static analysis and contemplates the forms best known and adopted by the industry in the application of software testing [59] (see Table II).

III. THE CHALLENGE OF TESTING

A. Software Quality

C. A. R. Hoare in the research “*How did software get so reliable without proof?*” conducted in 1996 states that it was reasonable to predict that the size and ambition of software products would be severely limited by the lack of reliability in their components. Estimates suggested, in its study, that professionally written programs may contain between one and ten correctable faults for every thousand lines of code; and any one software fault, in principle, can have a spectacular effect (or worse: a subtly misleading effect) on the behavior of the entire system [61].

Hoare found at the time that the software patch problem turned out to be far less serious than anticipated. An analysis by Mackenzie [62] showed that of several thousand deaths attributed to computer applications, only ten or so could be explained by software crashes: most due to a few cases of incorrect dosage calculations in radiation cancer treatment. Similarly, predictions of collapse due to the size of computer programs have been falsified by the continuous operation of

¹²If an action is visible to the environment (i.e. if it performs data retrieval or changes data), then we say it is observable. The order of execution of non-priority rules will make a difference in the order of appearance of observable actions.

¹³Although it is possible, as we can see later, to predict overflow using one of the many static analysis methods available.

Classe	Descrição
Unit Test	<i>The process of testing subprograms, subroutines, classes, or functional units within a program to verify that there are no programming flaws [60, p. 486].</i>
Integration Test	<i>Testing phase where the functional units are combined and tested as a group to assess whether they worked properly in the complete system [60, p. 235].</i>
System Testing	<i>Test conducted on multiple integrated systems to evaluate their ability to communicate with each other and achieve general and specific integration requirements [60, p. 545].</i>
Acceptance Test	<i>Testing of a system or functional unit generally performed by the buyer or user on-site after installation of the software to make sure that the contractual requirements have been met [60, p. 5].</i>

TABLE II: Classification of Dynamic Tests

real-time software systems now measured in tens of millions of lines of code and subject to thousands of updates per year.

In his review Hoare concludes that, despite appearances, modern software engineering practice owes much to the theoretical concepts and ideals of early research in this field; and that formalization and proof techniques have played an essential role in the validation and progress of research.

Hoare concludes that the main factors for the apparent success of the software are:

- **Management** - *The most dramatic advances in the delivery of reliable software are directly attributable to a wider recognition of the fact that the process of program development can be predicted, planned, managed, and controlled just as in any other branch of engineering.*
- **Test** - *Thorough testing is the cornerstone of reliability in quality assurance and control in modern production engineering. Tests are applied as early as possible throughout the production line. They are rigorously designed to maximize the probability of detecting failures and as quickly as possible.*
- **Debugging** - *The secret of successful testing is that it checks the quality of the process and methods by which the code was produced. But there is an entirely different and very common response to the discovery of a flaw by testing: simply fix it and get on with the job. This is known as debugging, by analogy with trying to get rid of a mosquito infestation by killing the ones that bite - much faster, cheaper and more satisfying than draining the swamps in which they breed.*
- **Excess Engineering** - *The concept of safety factor is very widespread in engineering. After calculating the worst case load on a beam, the civil engineer will try to build it at least twice as strong. In computing, a continuous drop in the price of storage and increased processing power has made it acceptable to add redundancies to reduce the risk of software failures and a smaller scale of damage. This leads to the same kind of over-engineering required by law for bridge construction; and it is extremely effective, although there is no clear way to measure it by a numerical factor.*
- **Programming Methodologies** - *Most of the measures described so far for achieving reliability in software are the same ones that have been proven equally effective in all engineering disciplines. But the best general techniques for management, quality control, and safety would be totally useless by themselves; they are only effective*

when there is a general understanding, a common conceptual framework and terminology for discussing the relationship between cause and effect, between action and consequence. Research in programming methodology has this goal: to establish a conceptual framework and a theoretical basis to assist in the systematic derivation and justification of each design decision by a rational and explicable line of reasoning.

B. Perceived Quality when Using Software

According to the [NIST \(National Institute of Standards and Technology\)](#) report, the estimated impact (in the United States) of inadequate software testing infrastructure is 859 billions dollars and the potential cost savings from feasible improvements is 822 billions dollars. Software users account for a larger share of the total costs of inadequate infrastructure (64 percent) compared to “viable” cost reductions (52 percent) because a large share of user costs are due to prevention activities. Whereas mitigation activities decrease proportionally to the decrease in the number of failures, prevention costs (such as redundant systems and investigating purchasing decisions) are likely to persist, even if only a few errors are expected. For software developers, the feasible cost savings are approximately 50 percent of the total costs of inadequate infrastructure. This reflects a more proportional decrease in testing effort as testing resources and tools improve [63].

If we add up everything from minor inconveniences in our daily lives to incalculable human and social damage from software failures, the perception we have may be quite different from that of Hoare in his study. This is because today the penetration of computerized systems in our lives, with its own challenges and opportunities due to the great convergence of connected systems, interoperability and massive distribution of information, can make the most insignificant failure from a mere annoyance (such as losing access to your favorite music playlist) to a catastrophe of global proportions (such as a widespread failure in a worldwide satellite communications system).

C. The Gap Between Industry and Scientific Advances

In 1996 Hoare noted that academic research gains in programming methodologies took up to 20 years to be adopted by industry as a sign of maturity and sanity - only in very specific areas and for a brief period would it be justified to apply the latest pure research advances to people’s everyday lives [61]. This mismatch also has the benefit of providing adequate

	Testers / Employees (millions)	Cost of inadequate testing infrastructure		Potential cost reduction with feasible improvements	
		Unit cost	Total cost (million US\$)	Unit cost (million US\$)	Total Cost (million US\$)
Developers	0,302	69,945	21.155	34,964	10,575
Users					
Industry	25,0	459	11.463	135	3,375
Services	74,1	362	26.858	112	8,299
Total			59,477		22,249

TABLE III: Estimated national impact in the US (adapted from [63])

Research	Type	Tests	Time	Defects
“A case study on pairwise testing application” [66]	<i>Ad-hoc</i>	14,041	20h	10
	<i>Pairwise</i>	68	4h	10
“A case study using testing technique for software as a service (SaaS)” [67]	<i>Manual</i>	159	6h	3
	<i>Pairwise</i>	17	1h	3

TABLE IV: Pairwise Application Research Results

planning of research and education as well as adequacy of the installed park in the industry. The result of not following this step is to adopt immature technologies and practices, with unpredictable and undesirable results, with no skilled labor available to apply it and make the necessary corrections when failures occur¹⁴.

Another consequence of not observing the maturity of cutting-edge research before its adoption in practice is the fact that, paradoxically, mature and effective technologies have not yet been adopted by industry, or when they are, they are isolated cases that cause astonishment when they present better results than those obtained with “state-of-the-art technologies”. As an example we cite the adoption of the *pairwise* technique for test generation. The mathematical theory behind this technique has been around since the 1960s (see *DESIGN, TESTING AND ESTIMATION IN COMPLEX EXPERIMENTATION. I. EXPANSIBLE AND CONTRACTIBLE FACTORIAL DESIGNS AND THE APPLICATION OF LINEAR PROGRAMMING TO COMBINATORIAL PROBLEMS* published in 1965 [64]), the application in software testing using *pairwise* was presented earlier this century (see *Combinatorial group testing and its applications* published in 2000 [65]). Recent research using these techniques (see Table IV) shows promising numbers¹⁵:

With results like this, it was expected that the adoption of the *Pairwise* technique to tests production in a cost-effective way would be more welcomed by the industry¹⁶.

IV. PROMISES OF FORMAL DEVELOPMENT

A. Model Driven Development

One of the most promising approaches to computer program development was *MDD (Model Drive Development)* and *MDA (Model Drive Architecture)*, where models are the primary

¹⁴If this scenario sounds like something that is happening in your industry, then maybe this is the reason.

¹⁵We are aware that this sampling is neither meaningful nor representative, but only illustrative from our point of view.

¹⁶Informally, in our contacts with software development practitioners and testing experts and discussions about the practice of *Pairwise* have ranged from ignorance of its existence to negative concepts and objections to its use as ineffective.

artifacts and the others, such as code, are generated from them [68]. The goal is to raise the level of abstraction, making software development closer to solving the requirements and problems outlined by its future users and making the developer’s life simpler and easier [69] and providing mainly automation of the process [70]. According to Yusuf, Chessell, and Gardner and Swithinbank, Chessell, Gardner, *et al.*, the advantages of using *MDD* are:

- Increased developer productivity - because of automation and focus on requirements analysis
- Ease of maintenance - many software was developed by specialists who left the organization at some point, and the technique would facilitate the evolution by retaining the knowledge of these specialists
- Legacy reuse - can make it easy and feasible to migrate old applications to new systems by applying the technique
- Adaptability - adding or modifying is made easy given the automation already in place
- Consistency - every application will strictly follow the pattern established by the tools
- Repetition - great return on investment if applied throughout an organization
- Improved communication with sponsors - models are easier to interpret than code
- Improved project communication - templates help to understand the system design and assist in the discussion about the system itself
- Domain knowledge capture - if there is sufficient documentation of the system, the organization’s knowledge is maintained
- Long-term asset - high-level models and abstractions of business solutions are immune to technological change
- Ability to postpone technology decisions - focus on solving business problems allows decisions on non-functional problems to be left for a more opportune time

1) *Problems with models*: The biggest problem with using models as the only source for software production is that trying to solve an organizational problem from conceptual abstractions larger than the machine languages used by computers to run programs implies a reduction of information [72, p. 90].

This information has to be supplanted by the *MDD* tool itself by means of ready-made patterns, or from the developer by means of extensions, and that leads, according to Hailpern and Tarr [69] to other problems:

- Redundancy - because of the widespread use of ready-made code examples
- Unbridled back and forth problems - to adjust the model to conform to another system or module
- Moving complexity elsewhere rather than reducing it, requiring even more specialization

2) *Future of MDD*: Standardization around *UML* and tool interoperability around the *XMI (XML Metadata Interchange)*[73] standard can lead the open source community to produce products that can leverage development using *MDD*. Tools such as the Eclipse Modeling Framework (see <https://www.eclipse.org/modeling/emf/>) is an example of technology with this kind of potential, however this leads us to another conclusion.

3) *Prospects*: Our view is that, the main barrier to the adoption of technologies like *MDD*, is how quickly this kind of solution becomes irrelevant.

This irrelevance happens as the application and use of information technologies and platforms evolve.

In the 1970s and 1980s, the adoption of *CASE (Computer Aided Software Engineering)* tools, which we can say were the precursors of *MDD* and *MDA*, was seen as a solution to the same problems we have listed above. At that time software development took place mainly on large computers, the *Mainframes*. But at the same time personal computers emerged, which at first were not seen as business tools, this soon became an untruth with the release of the IBM PC in 1981[74] and since then software development has moved from the older and more expensive platform (*Mainframes*) to the more modern and cheaper (*PCs (Personal Computers)*), and this became increasingly true with the adoption of local networks like Novell in 1979 [75] with over 500.000 computers installed in the world [76] at the time. This movement continued, but once again changed focus. In 1989 Tim Berners-Lee invented the World Wide Web, in 1993 we had the release of the Mosaic browser by *NCSA (National Center for Supercomputing Applications)*, and in 1994 we had Netscape Navigator created by the same developers, now in a private company of the same name. Since then the development has been turning to applications presented by the browsers but running on corporate servers on the Internet. In early 2007 Apple introduces the iPhone, and at the end of the following year Google introduces Android. Still supported by the basic Internet infrastructure, application development shifts focus once again to the new mobile platform. And these days, some technologies are on the threshold, or at least promise to be, of creating new platforms, and among them we can mention Bitcoin (announced in 2009), virtual reality (as used in airplane pilot training and introduced as a consumer product in the 1990s by computer game companies like Sega in 1991) and augmented reality (made popular in games like Pokémon Go in 2016) and finally the renaissance of Artificial Intelligence with the adoption of Machine Learning techniques.

This rapid evolution and shift of focus to different platforms, with different approaches that decisively impact the architecture of the systems, databases, operating systems, programming languages, forms of presentation, number of application layers, and different *APIs (Application Programming Interfaces)* employed to mediate an increasingly large and complex network of interconnected products and services makes it practically impossible to develop, train personnel, and make them productive in the employment of any technology with the nature of the *MDDs* tools, which end up being relegated only to the role of modeling, right at the initial requirements gathering phase, within a longer development life cycle and without fulfilling the promise of covering it completely that has been made since the 1970s and 1980s by the *CASE*[77] tools, and which, as we saw earlier in this introduction, often does not motivate software development professionals and decision makers to bear the cost and time required in their absorption and deployment.

V. CONCLUSIONS AND FUTURE WORK

If in one hand we have the promise of great advances and improvements in the quality of software products by applying techniques and tools developed by both academia and industry, despite the expected (and even desirable) delay between the development and adoption of these new technologies, we also have on the other hand the adoption of practices by the industry that make it difficult to incorporate certain mature technologies, or even to put them to the test, due to the lack of formalization that these practices prescribe in the name of agility in producing products quickly and meeting the desires of their customers.

Without the adoption of formal software development methods, it is not possible to continue and progress with the advanced quality methods and methodologies developed in academia.

The solution to this would be a back-and-forth approach, whereby by reverse engineering and starting from the source code of the computer programs, formal models are deduced and then complemented by the developers in order to produce the artifacts and inputs necessary for formal methods of quality verification and validation. Automation and adoption of standards are key to keeping costs within acceptable parameters for the industry.

This approach has its pros and cons. Using reverse engineering to produce formal models will cause loss of information¹⁷, and this and other problems to come are what we set out to address.

We intend to continue these studies with an analysis of the State of the Art in the conception and production of computer program tests, followed by ways of bringing together the methods and practices adopted by industry and the techniques developed by academia.

¹⁷In general, models have less information than the finished products that originated from them

ACRONYMS

API	Application Programming Interface - page: 6
CASE	Computer Aided Software Engineering - page: 6
CCS	Calculus of Communicating Systems - page: 1
CSP	Communicating Sequential Processes - page: 1
FSM	Finite State Machine - page: 2
IT	Information Technology - page: 1
LOTOS	Language Of Temporal Ordering Specification - page: 2
LTS	Labelled Transition Systems - page: 2
MDA	Model Drive Architecture - pages: 5, 6
MDD	Model Drive Development - pages: 5, 6
NCSA	National Center for Supercomputing Applications - page: 6
NIST	National Institute of Standards and Technology - page: 4
OCL	Object Constraint Language - page: 1
PC	Personal Computer - page: 6
TDD	Test Driven Development - pages: 1, 2
UML	Unified Modelling Language - pages: 1, 6
XMI	XML Metadata Interchange - page: 6
XML	Extensible Markup Language - pages: 6, 7

REFERENCES

- [1] K. Beck, M. Beedle, A. Van Bennekum, *et al.*, “*Manifesto for agile software development*,” 2001.
- [2] B. Ramesh, L. Cao, K. Mohan, and P. Xu, “*Can distributed software development be agile?*” *Communications of the ACM*, vol. 49, no. 10, pp. 41–46, 2006.
- [3] V. One, “*14th annual state of agile report*,” Online: <https://stateofagile.com>, 2020.
- [4] R. Hoda, J. Noble, and S. Marshall, “*How much is just enough? some documentation patterns on agile projects*,” in *Proceedings of the 15th European Conference on Pattern Languages of Programs*, 2010, pp. 1–13.
- [5] A. Cockburn and J. Highsmith, “*Agile software development, the people factor*,” *Computer*, vol. 34, no. 11, pp. 131–133, 2001.
- [6] D. G. Feitelson, E. Frachtenberg, and K. L. Beck, “*Development and deployment at facebook*,” *IEEE Internet Computing*, vol. 17, no. 4, pp. 8–17, 2013.
- [7] G. G. Claps, R. B. Svensson, and A. Aurum, “*On the journey to continuous deployment: Technical and social challenges along the way*,” *Information and Software technology*, vol. 57, pp. 21–31, 2015.
- [8] M. Hilton, T. Tunnell, K. Huang, D. Marinov, and D. Dig, “*Usage, costs, and benefits of continuous integration in open-source projects*,” in *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, IEEE, 2016, pp. 426–437.
- [9] M. Canat, N. P. Català, A. Jourkovski, S. Petrov, M. Wellme, and R. Lagerström, “*Enterprise architecture and agile development: Friends or foes?*” In *2018 IEEE 22nd International Enterprise Distributed Object Computing Workshop (EDOCW)*, IEEE, 2018, pp. 176–183.
- [10] K. Beck, “*Aim, fire [test-first coding]*,” *IEEE Software*, vol. 18, no. 5, pp. 87–89, 2001.
- [11] J. B. Goodenough and S. L. Gerhart, “*Toward a theory of test data selection*,” *IEEE Transactions on software Engineering*, no. 2, pp. 156–173, 1975.
- [12] T. S. Chow, “*Testing software design modeled by finite-state machines*,” *IEEE transactions on software engineering*, no. 3, pp. 178–187, 1978.
- [13] C. A. R. Hoare, “*An axiomatic basis for computer programming*,” *Communications of the ACM*, vol. 12, no. 10, pp. 576–580, 1969.
- [14] R. W. Floyd, “*Toward interactive design of correct programs*,” in *Readings in artificial intelligence and software engineering*, Elsevier, 1986, pp. 331–334.
- [15] J. B. Warmer and A. G. Kleppe, *The Object Constraint Language: Precise Modeling with UML*. Addison Wesley, 1999.
- [16] M.-C. Gaudel, “*Formal methods for software testing*,” in *2017 International Symposium on Theoretical Aspects of Software Engineering (TASE)*, IEEE, 2017, pp. 1–3.
- [17] B. Roscoe, “*The theory and practice of concurrency*,” 1998.
- [18] R. Milner, “*Lectures on a calculus for communicating systems*,” in *International Conference on Concurrency*, Springer, 1984, pp. 197–220.
- [19] E. Brinksma, “*An algebraic language for the specification of the temporal order of events in services and protocols*,” in *Proc. of the European Teleinformatics Conference, Varese, Italy*, 1983, pp. 533–542.
- [20] M. de Almeida Xavier, “*Definição e implementação do sistema de tipos da linguagem circus*,” M.S. thesis, Universidade Federal de Pernambuco, 2006.
- [21] M. L. Minsky, *Computation*. Prentice-Hall Englewood Cliffs, 1967.
- [22] E. M. Clarke, E. A. Emerson, and A. P. Sistla, “*Automatic verification of finite-state concurrent systems using temporal logic specifications*,” *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 8, no. 2, pp. 244–263, 1986.
- [23] C. A. Petri and W. Reisig, “*Petri net*,” *Scholarpedia*, vol. 3, no. 4, p. 6477, 2008.

- [24] B. George and L. Williams, "A structured experiment of test-driven development," *Information and Software Technology*, vol. 46, no. 5, pp. 337–342, 2004.
- [25] F. Shull, G. Melnik, B. Turhan, L. Layman, M. Diep, and H. Erdogmus, "What do we know about test-driven development?" *IEEE software*, vol. 27, no. 6, pp. 16–19, 2010.
- [26] M. Josefsson, "Making architectural design phase obsolete-tdd as a design method," in *Seminar course on SQA in Agile Software Development Helsinki University of Technology*, 2004.
- [27] L. Madeyski, "The impact of test-first programming on branch coverage and mutation score indicator of unit tests: An experiment," *Information and Software Technology*, vol. 52, no. 2, pp. 169–184, 2010.
- [28] D. Fucci, H. Erdogmus, B. Turhan, M. Oivo, and N. Juristo, "A dissection of the test-driven development process: Does it really matter to test-first or to test-last?" *IEEE Transactions on Software Engineering*, vol. 43, no. 7, pp. 597–614, 2016.
- [29] C. G. Hempel, "Studies in the logic of confirmation (i.)," *Mind*, vol. 54, no. 213, pp. 1–26, 1945.
- [30] G. ISO, "Information technology, open systems interconnection, conformance testing methodology and framework," *International Standard IS*, vol. 9646, 1991.
- [31] E. Dijkstra, "Structured programming," in *Classics in software engineering*, 1979, pp. 41–48.
- [32] P. W. Gaffney and J. W. Wooten, "Methods to ensure the standardization of fortran software. [pfort, dave, polish, and brnanl, for analysis and editing of codes, in fortran for pdp-10 and ibm 360 and 370]," May 1980.
- [33] A. Aiken, J. M. Hellerstein, and J. Widom, "Static analysis techniques for predicting the behavior of active database rules," *ACM Transactions on Database Systems (TODS)*, vol. 20, no. 1, pp. 3–41, 1995.
- [34] N. Ayewah, W. Pugh, D. Hovemeyer, J. D. Morgenthaler, and J. Penix, "Using static analysis to find bugs," *IEEE software*, vol. 25, no. 5, pp. 22–29, 2008.
- [35] R. P. Wilson and M. S. Lam, "Efficient context-sensitive pointer analysis for c programs," *ACM Sigplan Notices*, vol. 30, no. 6, pp. 1–12, 1995.
- [36] I. F. Darwin, *Checking C Programs with lint*. "O'Reilly Media, Inc.", 1988.
- [37] D. Evans, "Static detection of dynamic memory errors," *ACM SIGPLAN Notices*, vol. 31, no. 5, pp. 44–53, 1996.
- [38] D. Jackson, "Aspect: Detecting bugs with abstract dependences," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 4, no. 2, pp. 109–145, 1995.
- [39] D. L. Detlefs, "An overview of the extended static checking system," in *Proceedings of The First Workshop on Formal Methods in Software Practice*, Citeseer, 1996, pp. 1–9.
- [40] D. L. Detlefs, K. R. M. Leino, G. Nelson, and J. B. Saxe, "Extended static checking," 1998.
- [41] J. L. Jensen, M. E. Jørgensen, M. I. Schwartzbach, and N. Klarlund, "Automatic verification of pointer programs using monadic second-order logic," in *Proceedings of the ACM SIGPLAN 1997 conference on Programming language design and implementation*, 1997, pp. 226–234.
- [42] P. Li and B. Cui, "A comparative study on software vulnerability static analysis techniques and tools," in *2010 IEEE international conference on information theory and information security*, IEEE, 2010, pp. 521–524.
- [43] C. Hankin and D. Le Métayer, "Deriving algorithms from type inference systems: Application to strictness analysis," in *Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 1994, pp. 202–212.
- [44] L. D. Fosdick and L. J. Osterweil, "Data flow analysis in software reliability," *ACM Computing Surveys (CSUR)*, vol. 8, no. 3, pp. 305–330, 1976.
- [45] F. Hayes-Roth, "Rule-based systems," *Communications of the ACM*, vol. 28, no. 9, pp. 921–932, 1985.
- [46] R. S. Boyer, B. Elspas, and K. N. Levitt, "Select—a formal system for testing and debugging programs by symbolic execution," *ACM SigPlan Notices*, vol. 10, no. 6, pp. 234–245, 1975.
- [47] J. C. King, "Symbolic execution and program testing," *Communications of the ACM*, vol. 19, no. 7, pp. 385–394, 1976.
- [48] W. E. Howden, "Experiments with a symbolic evaluation system," in *Proceedings of the June 7-10, 1976, national computer conference and exposition*, 1976, pp. 899–908.
- [49] L. A. Clarke, "A program testing system," in *Proceedings of the 1976 annual conference*, 1976, pp. 488–491.
- [50] S. Abramsky and C. Hankin, *Abstract interpretation of declarative languages*. Prentice Hall Professional Technical Reference, 1987.
- [51] F. Nielson and N. Jones, "Abstract interpretation: A semantics-based tool for program analysis," *Handbook of logic in computer science*, vol. 4, pp. 527–636, 1994.
- [52] M. Davis, "The early history of automated deduction: Dedicated to the memory of hao wang," in *Handbook of Automated Reasoning*, Elsevier, 2001, pp. 3–15.
- [53] W. Bibel, "Early history and perspectives of automated deduction," in *Annual Conference on Artificial Intelligence*, Springer, 2007, pp. 2–18.
- [54] E. M. Clarke Jr, O. Grumberg, D. Kroening, D. Peled, and H. Veith, *Model checking*. MIT press, 2018.
- [55] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge, "Why don't software developers use static analysis tools to find bugs?" In *2013 35th International Conference on Software Engineering (ICSE)*, IEEE, 2013, pp. 672–681.
- [56] S. Panichella, V. Arnaoudova, M. Di Penta, and G. Antoniol, "Would static analysis tools help developers with code reviews?" In *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, IEEE, 2015, pp. 161–170.
- [57] F. Zampetti, S. Scalabrino, R. Oliveto, G. Canfora, and M. Di Penta, "How open source projects use static code analysis tools in continuous integration pipelines," in *2017 IEEE/ACM 14th International Conference on Min-*

- ing Software Repositories (MSR)*, IEEE, 2017, pp. 334–344.
- [58] M. Agnihotri and A. Chug, “A systematic literature survey of software metrics, code smells and refactoring techniques,” *Journal of Information Processing Systems*, vol. 16, no. 4, pp. 915–934, 2020.
- [59] G. J. Myers, T. Badgett, T. M. Thomas, and C. Sandler, *The art of software testing*. Wiley Online Library, 2004, vol. 2.
- [60] I. (O. for Standardization), *Iso/iec/ieee 24765: 2017 systems and software engineering-vocabulary*, 2017.
- [61] C. A. R. Hoare, “How did software get so reliable without proof?” In *International Symposium of Formal Methods Europe*, Springer, 1996, pp. 1–17.
- [62] D. MacKenzie, “Computer-related accidental death: An empirical exploration,” *Science and Public Policy*, vol. 21, no. 4, pp. 233–248, 1994.
- [63] S. Planning, “The economic impacts of inadequate infrastructure for software testing,” *National Institute of Standards and Technology*, 2002.
- [64] S. R. Webb, “Design, testing and estimation in complex experimentation. i. expansible and contractible factorial designs and the application of linear programming to combinatorial problems,” ROCKETDYNE CANOGA PARK CA, Tech. Rep., 1965.
- [65] D. Du, F. K. Hwang, and F. Hwang, *Combinatorial group testing and its applications*. World Scientific, 2000, vol. 12.
- [66] C. B. Monteiro, L. A. V. Dias, and A. M. da Cunha, “A case study on pairwise testing application,” in *2014 11th International Conference on Information Technology: New Generations*, IEEE, 2014, pp. 639–640.
- [67] A. C. da Silva, L. R. Correa, L. A. V. Dias, and A. M. da Cunha, “A case study using testing technique for software as a service (saas),” in *2015 12th International Conference on Information Technology: New Generations*, IEEE, 2015, pp. 761–762.
- [68] L. Yusuf, M. Chessel, and T. Gardner, “Implement model-driven development to increase the business value of your it system,” Retrieved January, vol. 29, p. 2008, 2006.
- [69] B. Hailpern and P. Tarr, “Model-driven development: The good, the bad, and the ugly,” *IBM systems journal*, vol. 45, no. 3, pp. 451–461, 2006.
- [70] R. Jacobs, *ARCAst with Ron Jacobs*, English. [Online]. Available: <https://channel9.msdn.com/Shows/ARCAst+with+Ron+Jacobs/ARCAst-5> (visited on 11/19/2020).
- [71] P. Swithinbank, M. Chessell, T. Gardner, *et al.*, *Patterns: Model-Driven Development Using IBM Rational Software Architect*. IBM, International Technical Support Organization, 2005.
- [72] S. K. Langer, *Feeling and form*. Routledge and Kegan Paul London, 1953, vol. 3.
- [73] O. M. Group, *XML Metadata Interchange*, English, Technology Standards Consortium, Jun. 2015. [Online]. Available: <https://www.omg.org/spec/XMI/About-XMI/>.
- [74] M. J. Miller, *Why the IBM PC had an Open Architecture*, English, News Site, publisher: Ziff Davis, Aug. 2011. [Online]. Available: <https://www.pcmag.com/archive/why-the-ibm-pc-had-an-open-architecture-286065>.
- [75] L. Proven, *How the clammy claws of Novell NetWare were torn from today’s networks*, English, News Site, publisher: Situation Publishing, Jul. 2013. [Online]. Available: https://www.theregister.com/2013/07/16/netware_4_anniversary/.
- [76] R. Payne and K. Manweiler, *CCIE: Cisco Certified Internetwork Expert Study Guide: Routing and Switching*. John Wiley & Sons, 2006.
- [77] V. J. Mercurio, B. F. Meyers, A. M. Nisbet, and G. Radin, “Ad/cycle strategy and architecture,” *IBM Systems Journal*, vol. 29, no. 2, pp. 170–188, 1990.