Syntactic-Semantic Incrementality for Agile Verification

Domenico Bianculli[a,*], Antonio Filieri[b], Carlo Ghezzi[c], Dino Mandrioli[c]

[a] *SnT Centre - University of Luxembourg, 4 rue Alphonse Weicker, L-2721, Luxembourg, Luxembourg*
[b] *Institute of Software Technology - University of Stuttgart, Universitätsstraße 38, D-70569, Stuttgart, Germany*
[c] *DEEPSE group - Dipartimento di Elettronica, Informazione e Bioingegneria - Politecnico di Milano, via Golgi 42, I-20133, Milano, Italy*

Abstract

Modern software systems are continuously evolving, often because systems requirements change over time. Responding to requirements changes is one of the principles of agile methodologies. In this paper we envision the seamless integration of automated verification techniques within agile methodologies, thanks to the support for incrementality. Incremental verification accommodates the changes that occur within the schedule of frequent releases of software agile processes. We propose a general approach to developing families of verifiers that can support incremental verification for different kinds of artifacts and properties. The proposed *syntactic-semantic* approach is rooted in operator precedence grammars and their support for incremental parsing. Incremental verification procedures are encoded as attribute grammars, whose incremental evaluation goes hand in hand with incremental parsing.

Keywords: Agile development, formal verification, operator-precedence grammars, parsing, attribute grammars, incremental algorithms.

This article was written to celebrate Paul Klint's 65th birthday; it honors his life-long passionate work on languages and grammars and their profound influence on software engineering.

1. Introduction

Modern software systems are embedded in an increasingly dynamic and open world, which is a source of continuous change [1]. The requirements the system is expected to satisfy change because the goals one tries to achieve through the system change over time. The domain assumptions upon which the software is built may also change, because of changes occurring in the surrounding environment. The consequence of these exogenous changes is that the software system is also required to change in order to fulfill its goals. This phenomenon is traditionally known as *software evolution*. Software engineering has long been concerned with supporting software evolution [2].

Software evolution should be supported by specific methods and tools. The systematic design of a software system, in fact, goes through a number of steps, which generate certain artifacts. Typically, high-level models are developed, analyzed, and transformed into increasingly more detailed descriptions and verified against certain properties. Eventually, an executable form is generated and deployed for execution. A change in the requirements or in the domain assumptions upon which the software depends implies that the development process needs to be revisited to accommodate changes. A naive approach would consist of viewing the changes as defining a new system and redoing the whole process from scratch. An incremental and iterative approach instead tries to control and manage the way changes occur, reusing the results of previous steps as much as possible, and focusing on the effects of changes.

Responding to software changes has been embraced as one of the principles at the basis of *agile development processes* [3]. An agile software development process is characterized by fast development iterations and allowance

*Corresponding author
Email addresses: domenico.bianculli@uni.lu (Domenico Bianculli), antonio.filieri@informatik.uni-stuttgart.de (Antonio Filieri), carlo.ghezzi@polimi.it (Carlo Ghezzi), dino.mandrioli@polimi.it (Dino Mandrioli)

for requirements evolution through interaction with the customers [4]. In the last decade, the adoption of these processes both in small companies and in large enterprises has been constantly increasing, until they became mainstream development approaches [5].

There is a cultural divide [6] between agile approaches and certain approaches that are instead extensively studied by research in software engineering, such as formal modeling, formal verification, and formal methods in general. These are in fact often viewed as disconnected from the final product that needs to be delivered, i.e., the code. Formal verification, in particular, is seen as inflexible, difficult and time-consuming to apply, and thus not immediately applicable in the context of agile software development processes. Formal models and verification — despite having become well-established and effective in the restricted case of mission- and safety-critical software systems development — are viewed as impediments to agile development rather than aids. At the same time, agile approaches have earned the dubious reputation of rapidly putting together software at the expense of quality. Nevertheless, there is a clear trend (see, for example, the recent editions of the *FM+AM* [7] and *FormSERA* [8] workshops), to integrate formal and agile methods and get the best of both worlds.

We envision here the future generation of software development processes based on agile methodologies where automated verification techniques are seamlessly and fully integrated in the development process, in much the same way as today testing is integrated through *test-driven development* (TDD). Two scientific advances justify this vision: the progress made in the area of *model-driven development* [9] and the progress made in *formal software verification* [10]. To progress in our long-term vision, however, further advances in research are needed. In particular, in this paper we focus on a research agenda aiming at blending formal verification into agile development.

The point we wish to make is that further research is needed to develop methods and tools that can make formal verification incremental, so that it can fit naturally within the cycle of incremental software releases that characterizes agile methodologies. Verification procedures should be executed incrementally, both to support reasoning about changes and to ensure fast turnaround, compatible with the tight schedule of the frequent software releases prescribed by agile processes. Verification agility may be pushed to its extreme so that it can be brought to run time to support self-adaptation, as changes are detected that would lead to requirements violations [11].

The idea that verification methods should be incremental to effectively deal with changes is not new [12]. Although various attempts have also been made to make specific verification approaches incremental, to the best of our knowledge all of these attempts are based on ad-hoc techniques (see [13] and also the discussion at the end of section 4). Here instead we aim at a generic approach that can be applied in principle to any artifact to verify and any property language in which we want to specify the requirements.

In this paper, we present a vision to achieve incremental verification by means of a *syntactic-semantic* approach, which exploits a particular class of grammars and their support for incremental parsing. In our approach, incremental verification procedures are encoded as attribute grammars.

The rest of the paper is organized as follows. Section 2 sketches the theoretical underpinnings at the basis of our approach. Our idea for agile, incremental verification is then illustrated in Section 3. Section 4 situates our approach with respect to the state of the art. Section 5 presents an outlook on future research.

2. Syntactic-Semantic Incremental Verification

We propose a general framework to define a variety of verification procedures, which are automatically enhanced with incrementality by the framework itself. The framework follows a *syntactic-semantic* approach [13]: it assumes that the software artifact to be verified has a syntactic structure described by a formal grammar, and that the verification procedure is encoded as the computation of semantic attributes [14], associated with the grammar[1] and evaluated by traversing the syntax tree of the artifact to verify. Verification procedures can then be carried out hand-in-hand with the parsing of the analyzed artifact. Since attribute synthesis is Turing complete [15], it allows for virtually any verification algorithm to be formalized through it.

[1] In this paper we assume grammars to have only synthesized attributes. Nonetheless, it can be proved that these grammars are semantically complete, since any inherited attribute can be translated into a (set of) synthesized attribute(s) reproducing the same information [14].

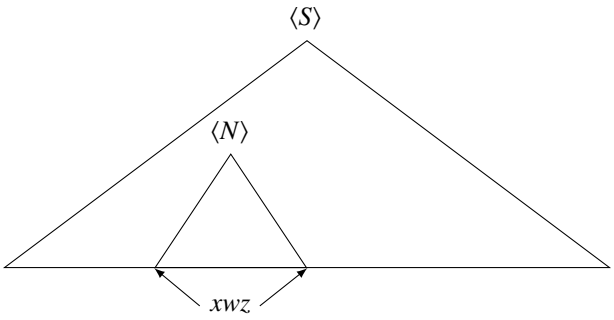

Figure 1: Syntax tree rooted in the axiom $\langle S \rangle$, with a subtree rooted in the non-terminal $\langle N \rangle$ generating the string xwz

The Theoretical Underpinnings

The theoretical underpinnings of this methodology include a strict subset of context-free grammars (CFGs)—known as *Operator Precedence Grammars* (OPGs) [16]—augmented with attributes, according to Knuth's attribute grammars. The main reason for the choice of OPGs is that, unlike other more widely used grammars that support deterministic parsing, they enjoy the *locality property*, i.e., the possibility of starting the parsing from any arbitrary point of the sentence to be analyzed, independent of the context within which the sentence is located.

Consider a scenario where, after having built a syntax tree for a given input program (i.e., a certain input sentence), one or more parts of the program are changed: thanks to the locality property only the changes should be re-parsed. Afterwards, the new local parse subtrees should be merged with the global parse tree using a suitable criterion. We say that the *matching condition* is satisfied when, after having parsed a substring, it is possible to identify the correct nesting point of its parse subtree within the global one [17]. For example, consider the generic syntax tree depicted in Fig. 1, in which the non-terminal $\langle N \rangle$ generates the string xwz. Suppose that substring w is replaced by a new string w'. The OPG parsing algorithm can restart parsing from the substring w' and its context $x - z$, regardless of the rest of the input. The parsing algorithm will apply subsequent reductions until the derivation $\langle N \rangle \Rightarrow xw'z$ is found: this satisfies the matching condition and the old subtree rooted in $\langle N \rangle$ is replaced with the new one[2].

Since in a bottom-up parser semantic actions are only performed during a reduction, the recomputation of semantic attributes proceeds hand-in-hand with the re-parsing of the modified substring. Continuing the example described above, assume non-terminal $\langle N \rangle$, root of the newly re-parsed subtree, has an attribute α_N. Upon reparsing of the new string w', a new subtree will be rooted in $\langle N \rangle$. Two situations may occur related to the computation of α_N:

1. The α_N attribute associated with the new subtree rooted in $\langle N \rangle$ has the same value as before the change. In this case, all the remaining attributes in the rest of the tree will not be affected, and no further analysis is needed.

2. Despite the syntactic matching, the new value of α_N is different from the one it had before the change. In this case, as suggested by Fig. 2, only the attributes on the path from $\langle N \rangle$ to the root $\langle S \rangle$ (e.g., $\alpha_M, \alpha_K, \alpha_S$) can change and thus need to be recomputed. The values of the other attributes not on the path from $\langle N \rangle$ to the root (e.g., α_P and α_Q) do not change and thus there is no need to recompute them.

The locality property also supports parallel parsing, possibly to be exploited in a natural combination with incrementality. In case of multiple changes to the input string, if the subtrees affected by the changes are disjoint, i.e. their contexts do not overlap, as depicted in Fig. 3, the tasks of parsing the two new substrings $xw'z$ and $yv's$ can be performed by two processes and completed in a totally independent way. If the two subtrees share at least one node, the matching condition is not satisfied, and the two partial parsings have to be merged together. The two changes can be re-parsed separately until the respective subtrees share at least one node. Once the two processes executing the parsing are about to apply a reduction involving (at least) one node shared by both subtrees, the control is passed to only one of the processes, which then completes the parsing by itself[3].

Let us remark that the locality property has a price in terms of generative power. LR grammars, traditionally

[2]Notice that the algorithm detects automatically the shortest context of w' that satisfies the matching condition.

[3]This simple strategy can be significantly enhanced by means of a more precise identification of the nodes that effectively need to be changed, e.g. by jointly applying $LR \cap RL$ techniques [18].

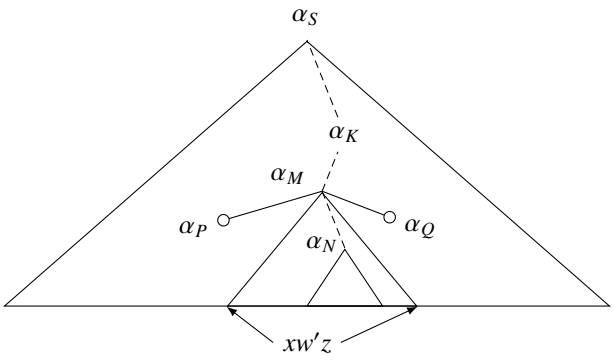

Figure 2: Incremental evaluation of semantic attributes

used to describe and parse programming languages, do not enjoy it, though they can generate all the deterministic languages. OPGs cannot; this limitation, however, is more of theoretical interest than of real practical impact. Large parts of the grammars of many computer languages are operator precedence [19, p. 271]; for instance, in the past complete OPGs have been given for Prolog [20] and Algol 68 [21]. Moreover, in many practical cases one can obtain an OPG by minor adjustments to a non operator-precedence grammar [16].

3. Incremental Verification meets Agile Development

The main outcome of the syntactic-semantic approach sketched in the previous section will take the form of a framework for generating incremental verification procedures whose main characteristics are:

- *Generality*: any verification procedure can in principle be formalized in the framework, although the benefits of incrementality may differ among different verification strategies of the same properties. One of the main goals of our work will be to identify convenient guidelines for the users.
- *Flexibility*: the syntax to describe the artifacts to be analyzed has to be formalized through an OPG. No further constraints are in place, allowing both for general purpose and for domain-specific programming languages, as well as automata-based definitions, to be analyzed.
- *Unification*: the same methodology and framework can be applied for the verification of both functional and non-functional requirements. As an example of the former class, we made an initial experiment with this approach to support incremental verification of *qualitative* program reachability properties. As an example of the latter, we made another experiment with incremental verification of *quantitative* (probabilistic) reliability properties of composite Web services defined by a workflow language. The ability to reason about both quantitative and qualitative properties

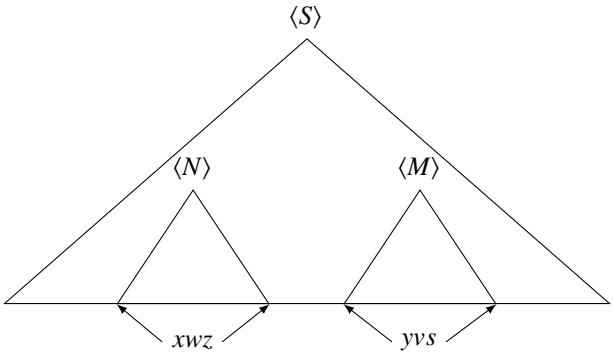

Figure 3: Parallel incremental update of a parse tree

opens new avenues for multi-objective and trade-off analysis that can deal with both functional and non-functional requirements.

The framework we envision fits agile, iterative software development processes, since it exploits incrementality to introduce *verification-driven development* methodologies in the context of such processes. In analogy with test-driven development, requirements would be formalized as verification problems (encompassing both functional and non-functional aspects) and development would proceed hand-in-hand with verification. Complementing modern agile techniques (such as TDD) with more accurate, general, and powerful verification techniques will further improve the quality of the software being developed.

As mentioned, we aim at a *general* methodology for incremental verification and a reusable framework to support it. Instead of adapting a specific verification procedure to make it incremental, our approach provides a general mechanism, which is neutral both with respect to the artifact (provided it can be described via an OPG) and the property (provided it is specified via attributes) to be verified. The framework itself will then automatically generate ready-to-use, incremental verification procedures.

Our approach can be described by the workflow depicted in Fig. 4. First, *domain experts* use their domain knowledge (e.g., defining reliability estimations by means of stochastic processes) to help *verification engineers* in defining a *verification procedure plugin*, which consists of an OPG (for the artifact language to support) and the associated attribute grammar scheme corresponding to the verification procedure one wants to implement. The verification procedure plugin is then integrated with the *backbone*, to generate an incremental syntactic-semantic verification tool. This tool takes as input the software artifacts to analyze and the requirements against which the analysis has to be performed. Requirements are assumed to be formalized in a convenient notation/format (e.g., a temporal logic or an automaton-based description) for the verification procedure plugin. The incremental verification tool is then used by developers to verify software artifacts, as part of the quality assurance activities of the project. The output of the tool is then used to correct software artifacts and/or refine requirements in the next development iteration. To popularize the adoption of our approach, we plan to integrate it within an agile development environment.

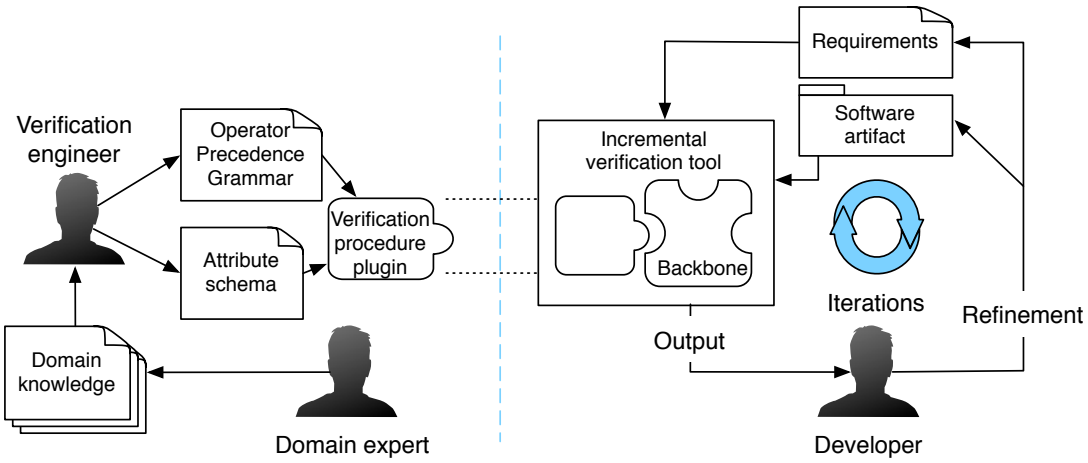

Figure 4: Workflow established by the incremental verification framework

4. State of the Art and Related Work

Different methodologies have been proposed in the literature to support incremental verification techniques [13]. *Assume-guarantee* is the grounding paradigm for most incremental verification techniques. This paradigm describes systems as a collection of cooperating modules, each of which has to guarantee certain properties [22]. The verification methods based on this paradigm are said to be compositional, since they allow reasoning about each module separately and then deducing properties about their integration. If the effect of a change can be localized inside the boundary of a module, the other modules are not affected, and their verification does not need to be redone [23]. Approaches

based on assume-guarantee have been proposed not only for qualitative verification, but also for quantitative analysis of models [24]. The main limitations of assume-guarantee are due to its reliance on the modular structure of the system. It does not help in dealing with effects of changes that cross-cut the modular structure. Furthermore, its effectiveness may be limited by the relatively coarse grain of modules. Our approach instead leverages the natural system decomposition encoded in a parse tree, which is available at any granularity level (e.g., from a single statement to a block structure). It does not require any a priori decomposition into a modular structure to be performed by designers.

Closely related to assume-guarantee are the approaches explored in [25, 26, 27]. Intuitively, this line of work too assumes a predefined modularization of the system. Typically, it assumes that a part of the system is given, which interacts with other parts that are currently unknown and unspecified. In these approaches one starts from the global property the entire system has to satisfy, and breaks this property down into the properties that need to be satisfied by the yet unknown and unspecified parts. Such properties become proof obligations for the parts, whose verification will ensure satisfaction of the global property. This approach has been studied in the context of workflows that use external services, labelled transition systems where certain states are left unspecified, and Statecharts where a state is assumed to be a placeholder for a Statechart that can later be defined to detail it.

Parametric analysis is another enabling methodology for incremental verification, mostly used for the analysis of quantitative properties [28, 29, 11, 30]. The core idea is to verify certain properties for a model allowing for a certain degree of uncertainty to be accounted for by replacing some actual model parameters with symbolic placeholders. The result of parametric analysis is an expression having as unknowns those model parameters. If a change in the model can be conveniently reflected by a new assignment to parameters, it is possible to avoid re-executing the verification procedure by re-evaluating the symbolic expression. Parametric analysis requires anticipated knowledge of both the global structure of the system and the location of possible changes. Furthermore, it requires an ability to cast changeable information as model parameters. For this reason it does not quite fit agile development methods where the global software structure will only *emerge* through the various development iterations. The approach we advocate here is instead more general and supports any kind of change.

Regression model-checking refers to the enhancement of model checking procedures by tracing back to the already explored space-state and assessing which parts of it have been affected by a change (see [31, 32, 33, 34, 35]). This optimization may possibly reduce the state space to be explored. However, these approaches tie incrementality to the low-level details of the verification procedure, while our approach supports incrementality at a higher level, which does not depend on the algorithm and data structures defined in the grammar attributes. Hence, it represents a general approach that could also be used to recast existing verification techniques into an incremental framework. Similar considerations are valid for frameworks for the incrementalization of algorithms for specific program analysis tasks (see [36],[37]), based on *memoization* and change propagation.

In summary, the approaches based on compositional reasoning (e.g, *assume-guarantee*) and on a change anticipation (e.g., *parametric analysis*) are based on an a-priori *design for change* methodology, which assumes some knowledge of the system's parts that are most likely subject to future evolution and tries to encapsulate them within well-defined borderlines. Our approach, instead, does not make any hypothesis on where changes will occur during system's life: it simply evaluates a posteriori their scope within system's structure as formalized by the syntax tree and the dependencies among attributes. This should be particularly beneficial in most modern systems that evolve in a fairly unpredictable way, often without a unique design responsibility. Furthermore, with respect to existing approaches for (verification) algorithms incrementalization, our approach supports general and pre-defined mechanisms, embedded in the incremental parsing and attribute evaluation procedures.

The approach advocated by the this paper is rooted in earlier approaches to syntax-driven incrementality (such as past work by some of the authors on incremental parsing [18]) and to incremental attribute grammars evaluation [38].

The main theme of this paper is that incremental verification can become the cornerstone of future agile processes based on verification-driven development. Its motivation comes from past and current work by some of the authors in the area of self-adaptive software, which led to a run-time software adaptation approach that requires verification procedures to be efficiently executed at run time. The parametric approach for quantitative probabilistic verification presented in [11, 30] was developed precisely in this context. More generally, the work on verification at run time led to rethinking the role of incremental verification during development, evolution, and operation.

5. Outlook

Techniques for automated software verification are of crucial importance to assure development of applications that dependably satisfy their requirements. Wide and generalized adoption of such verification techniques within agile development strategies is currently hindered by the conflict between the need for fast development iterations and continuous refinement of software on the one side, and on the other side the performance of most formal verification tools, which do not provide specific support for incremental analysis. In this paper we outlined our idea for achieving incremental verification and plugging it into agile development processes, exploiting a syntactic-semantic approach.

By combining the best of both worlds, next generation software could be developed according to the principles of agile and iterative methodologies, which fit the evolving nature of software quite well, while retaining the benefits of rigor, precision, and dependability guaranteed by formal verification. As briefly mentioned, incremental verification approaches can be also applied in situations where changes occur at run time and the software itself is responsible for reacting to them in a timely and self-managed manner. In this setting, verification is often subject to severe time constraints. Because its outcomes are used to decide whether changes may lead to requirements violations and thus an adaptive reconfiguration has to be triggered to prevent such violations, its execution has to be extremely efficient. Conventional verification techniques are too heavy in terms of computational complexity to be applied on-line. But normally changes are small and local, and here is where incrementality can provide the breakthrough that allows for run-time system evolution, with minimal or even null service discontinuity. In addition, they do not support reasoning about the effect of changes, which instead is a very relevant information to be provided to the designers.

To support the vision of incremental verification we presented here, it is necessary to make progress in two main directions. One consists of setting up a generic incremental parsing and incremental attribute evaluation environment that may be used to generate incremental verifiers for different types of artifacts and property languages. The other must focus on challenging the generality and the expected efficiency gains from the approach. As for the latter issue, we plan to work on the encoding of a representative number of state of the art verification procedures for mainstream programming languages in terms of attribute grammars in view of their incremental evaluation. This step will require rethinking the verification algorithms and data structures. A preliminary report on the encoding of two verification procedures (model checking and reliability analysis) for a toy language can be found in [39]. Furthermore, we will investigate how existing optimizations for verification procedures can be introduced in the encoding of the procedure, without interfering with the incremental approach. However, also the optimization techniques will have to be redefined specifically in terms of attribute grammar evaluation. Moreover, we will perform a quantitative assessment of the efficiency gains of the incremental evaluation with respect to complete re-evaluations, using realistic case studies.

Two potential problems need to be scrutinized quite carefully:

1. Transforming existing grammars into an OPG form. As said in section 2, in many practical cases one can obtain an OPG by minor adjustments to a non operator-precedence grammar [16]. However, this transformation (especially when automated) might reduce the readability of the grammar and this could have an impact on the definition of attribute schemas[4].

2. Expressing verification procedures as attribute grammars. Developers will be likely unfamiliar with this programming paradigm. In addition, reuse of existing verification algorithms could be more or less straightforward and/or effective in this context.

These problems, however, are typically carried out once for all at design time, possibly in cooperation with domain experts. When the system is in operation, developers should only care about applying the changes and automatically (and incrementally) verifying their effects.

6. Conclusions

Syntax-driven approaches have proved in the past to provide extremely flexible and versatile support to tool development. The research we presented here aims at providing syntax-driven support for generating incremental formal

[4]For instance, in our early experience on this activity, we could obtain an OPG for JSON with minimal effort; one for Lua took a few days, with a considerable increase in the size of the grammar; JavaScript required more effort. We are presently working on making this activity more systematic and general.

verification tools for a variety of notations in which the artifacts to be verified can be encoded and property languages in which the desired requirements to be met by the artifacts can be expressed. We assume the syntactic structure of artifacts to be described by operator-precedence grammars and verification procedures to be expressed as semantic attributes. Incremental parsing and attribute evaluation algorithms will support incremental verification.

We envision incremental verification as a possible cornerstone of new generations of agile environments that will support the development, evolution, and run-time operation of future software.

Acknowledgements

This work has been partially supported by the European Community under the IDEAS-ERC grant agreement no. 227977-SMScom and by the National Research Fund, Luxembourg (FNR/P10/03).

References

[1] L. Baresi, E. Di Nitto, C. Ghezzi, Toward open-world software: Issues and challenges, IEEE Computer 39 (10) (2006) 36–43.
[2] M. Lehman, Programs, life cycles, and laws of software evolution, Proceedings of the IEEE 68 (9) (1980) 1060 – 1076.
[3] K. Beck, al., Manifesto for agile software development, `http://agilemanifesto.org/` (2001).
[4] C. Larman, V. Basili, Iterative and incremental developments: a brief history, Computer 36 (6) (2003) 47–56.
[5] A. Begel, N. Nagappan, Usage and perceptions of agile software development in an industrial context: An exploratory study, in: Proc. of ESEM'07, IEEE Computer Society, 2007, pp. 255–264.
[6] S. Black, P. P. Boca, J. P. Bowen, J. Gorman, M. Hinchey, Formal versus agile: Survival of the fittest, Computer 42 (9) (2009) 37–45.
[7] S. Gruner, B. Rumpe (Eds.), FM+AM 2010 - Second International Workshop on Formal Methods and Agile Methods, 17 September 2010, Pisa (Italy), Vol. 179 of LNI, GI, 2010.
[8] S. Gruner, B. Rumpe, Formsera workshop on formal methods in software engineering rigorous and agile approaches, SIGSOFT Softw. Eng. Notes 37 (6) (2012) 28–30.
[9] R. France, B. Rumpe, Model-driven development of complex software: A research roadmap, in: Proc. of FOSE 2007, IEEE Computer Society, 2007, pp. 37–54.
[10] M. B. Dwyer, J. Hatcliff, Robby, C. S. Păsăreanu, W. Visser, Formal Software Analysis Emerging Trends in Software Model Checking, in: Proc. of FOSE 2007, IEEE Computer Society, 2007, pp. 120–136.
[11] A. Filieri, C. Ghezzi, G. Tamburrelli, Run-time efficient probabilistic model checking, in: Proc. of ICSE 2011, ACM, 2011, pp. 341–350.
[12] P. Sistla, Hybrid and incremental model-checking techniques, ACM Comput. Surv. 28 (4es) (2006).
[13] C. Ghezzi, Evolution, adaptation, and the quest for incrementality, in: Proc. of the 17th Monterey Workshop, Vol. 7539 of LNCS, Springer, 2012, pp. 369–379.
[14] D. Knuth, Semantics of context-free languages, Mathematical systems theory 2 (2) (1968) 127–145.
[15] D. Milton, Syntactic specification and analysis with attributed grammars, Ph.D. thesis, University of Wisconsin-Madison (1977).
[16] R. W. Floyd, Syntactic analysis and operator precedence, J. ACM 10 (1963) 316–333.
[17] M. J. Fischer, Some properties of precedence languages, in: Proc. of STOC '69, ACM, 1969, pp. 181–190.
[18] C. Ghezzi, D. Mandrioli, Incremental parsing, ACM Trans. Program. Lang. Syst. 1 (1) (1979) 58–70.
[19] D. Grune, C. J. H. Jacobs, Parsing Techniques - a practical guide, 2nd Edition, Springer, 2008.
[20] K. de Bosschere, An operator precedence parser for standard Prolog text, Softw. Pract. Exper. 26 (7) (1996) 763–779.
[21] L. G. L. T. Meertens, J. C. van Vliet, An Operator-Priority Grammar For Algol 68+, CWI Technical Report IW 173/81, Stichting Mathematisch Centrum (1981).
[22] C. B. Jones, Tentative steps toward a development method for interfering programs, ACM Trans. Program. Lang. Syst. 5 (4) (1983) 596–619.
[23] J. M. Cobleigh, D. Giannakopoulou, C. S. Păsăreanu, Learning assumptions for compositional verification, in: Proc. of TACAS 2003, Vol. 2619 of LNCS, Springer, 2003, pp. 331–346.
[24] M. Kwiatkowska, G. Norman, D. Parker, H. Qu, Assume-guarantee verification for probabilistic systems, in: Proc. of TACAS 2010, Vol. 6015 of LNCS, Springer, 2010, pp. 23–37.
[25] D. Bianculli, D. Giannakopoulou, C. S. Păsăreanu, Interface decomposition for service compositions, in: Proc. of ICSE 2011, ACM, 2011, pp. 501–510.
[26] A. Molzam Sharifloo, P. Spoletini, LOVER: Light-weight fOrmal Verification of adaptivE systems at Run time, in: Proc. of FACS 2012, Vol. 7684 of LNCS, Springer, 2012, pp. 170–187.
[27] C. Ghezzi, C. Menghi, A. Molzam Sharifloo, P. Spoletini, On Requirements Verification for Model Refinments, in: Proc. of RE 2013, IEEE Computer Society, 2013, pp. 62–71.
[28] C. Daws, Symbolic and Parametric Model Checking of Discrete-Time Markov chains, in: Proc. of ICTAC 2004, Vol. 3407 of LNCS, Springer, 2005, pp. 280–294.
[29] E. Hahn, H. Hermanns, B. Wachter, L. Zhang, PARAM: A Model Checker for Parametric Markov Models, in: Proc. of CAV 2010, Vol. 6174 of LNCS, Springer, 2010, pp. 660–664.
[30] A. Filieri, C. Ghezzi, Further steps towards efficient runtime verification: Handling probabilistic cost models, in: Proc. of FormSERA, IEEE Computer Society, 2012, pp. 2–8.
[31] O. V. Sokolsky, S. A. Smolka, Incremental model checking in the modal mu-calculus, in: Proc. of CAV 1994, Vol. 818 of LNCS, Springer, 1994, pp. 351–363.

[32] G. Yang, M. Dwyer, G. Rothermel, Regression model checking, in: Proc. of ICSM 2009, IEEE Computer Society, 2009, pp. 115–124.
[33] T. A. Henzinger, R. Jhala, R. Majumdar, M. A. Sanvido, Extreme model checking, in: Verification: Theory and Practice, Vol. 2772 of LNCS, Springer, 2004, pp. 180–181.
[34] S. Lauterburg, A. Sobeih, D. Marinov, M. Viswanathan, Incremental state-space exploration for programs with dynamically allocated data, in: Proc. ICSE 2008, ACM, 2008, pp. 291–300.
[35] S. Krishnamurthi, K. Fisler, Foundations of incremental aspect model-checking, ACM Trans. Softw. Eng. Methodol. 16 (2) (2007) Article 7.
[36] A. Shankar, R. Bodík, DITTO: automatic incrementalization of data structure invariant checks, in: Proc. of PLDI, ACM, 2007, pp. 310–319.
[37] C. Conway, K. Namjoshi, D. Dams, S. Edwards, Incremental algorithms for inter-procedural analysis of safety properties, in: Proc. of CAV 2005, Vol. 3576 of LNCS, Springer, 2005, pp. 387–400.
[38] A. Demers, T. Reps, T. Teitelbaum, Incremental evaluation for attribute grammars with application to syntax-directed editors, in: Proc. of POPL '81, ACM, 1981, pp. 105–116.
[39] D. Bianculli, A. Filieri, C. Ghezzi, D. Mandrioli, A syntactic-semantic approach to incremental verification, `http://arxiv.org/abs/1304.8034` (2013).