SCIENCE CHINA
Information Sciences

• PERSPECTIVE •

May 2018, Vol. 61 056101:1–056101:3
https://doi.org/10.1007/s11432-017-9355-3

Can big data bring a breakthrough for software automation?

Hong MEI[1,2*] & Lu ZHANG[1]

[1]*Key Laboratory on High-Confidence Software Technologies (Ministry of Education), Peking University, Beijing* 100871, *China;*
[2]*Beijing Institute of Technology, Beijing* 100081, *China*

Received 15 December 2017/Accepted 15 January 2018/Published online 9 April 2018

Citation Mei H, Zhang L. Can big data bring a breakthrough for software automation? Sci China Inf Sci, 2018, 61(5): 056101, https://doi.org/10.1007/s11432-017-9355-3

Software automation [1] aims to automatically generate computer programs from formal or informal requirements. Since it may release programmers from tedious programming tasks, software automation has long been a dream of computer scientists. Practically, since software systems constantly evolve during their life cycles, software automation should cover all development activities related to both generating new code and changing existing code. Compilers for high-level programming languages (e.g., C and FORTRAN) can be viewed as pioneer work in the field of software automation. With compilers, programs written in high-level programming languages can be automatically transformed into their executable forms using some transformation rules. However, to realize software automation in the modern sense, where software systems written in high-level programming languages need to be automatically generated based on their requirements, three major challenges need to be further addressed: informality, non-operationality, and incompleteness.

• **Informality.** Instead of representing requirements for computers to process (e.g., a formal language), humans tend to represent requirements in a manner for humans to process (e.g., a natural language). To address this challenge, researchers have investigated various specification languages [2] (which can be either formal, semi-formal, or graphical) to provide a compromise. However, informality remains challenging because it implies that computers should understand natural languages in an accurate manner.

• **Non-operationality.** Instead of describing "how to do" in requirements, humans tend to describe only "what to do". To address this challenge, researchers have investigated various declarative languages (e.g., functional languages [3]), where developers can describe only "what to do" requirements and a compiler or an interpreter automatically maps them to their corresponding "how to do" requirements. However, declarative languages can manage only a limited set of such "what to do" requirements. To accommodate a broader scope of "what to do" requirements, a search procedure to synthesize programs satisfying the "what to do" requirements is needed. This kind of program synthesis is very difficult because it needs to search an infinite program space.

• **Incompleteness.** Instead of describing the full set of requirements, humans tend to explicitly provide a small subset of the requirements, keeping the remaining requirements latent. To address this challenge, researchers have investigated various domain-specific languages [4] for mature domains, where software systems differ from each other on a well-defined set of variation points. Thus, developers can use a domain-specific language to concisely describe a target system, thus alleviating the situation. The difficulty of fully ad-

* Corresponding author (email: meih@pku.edu.cn)

dressing this challenge is that the large number of unknown variation points in general domains poses an intrinsic barrier for humans to design and implement suitable domain-specific languages.

This article proposes that the vast accumulation of source code and related documentation (i.e., big data in software development) may shed some new insights into software automation. Here, big data in software development at least include software code bases, software revisions, software documents, software issues in issue tracking systems, and development-related emails among developers. Similar to other types of big data, these data are also accumulating at a rapid pace, although the absolute volume is far smaller than that of some typical kinds of big data (e.g., video data). However, due to their complex structures, efficient processing of these data is already a challenge.

For informality, the parallel relation between source code and its functionality descriptions in natural languages provides an opportunity to learn how to map descriptions in natural languages to source code. For non-operationality, since these data may generally characterize a space of existing software, confining the search space within this software space may both accelerate the search procedure and help produce human readable code. For incompleteness, since different software systems may share some common functionalities (which may not be specified formally and/or in an explicit form), these common functionalities may provide an opportunity for identifying latent requirements and their implementations in existing code.

Analog. By considering software automation as transforming requirement description into source code, software automation can be viewed as an analog to machine translation [5]. Although various rule-based approaches have been intensively investigated, the vast accumulation of parallel natural language corpora makes data-driven machine translation competitive or even more effective.

For software automation, what has been done in data-driven machine translation is mainly suitable to address only informality. To address the other two challenges, invention of new methodologies and/or techniques becomes unavoidable, because the other two challenges do not essentially involve pure translation but more or less involve creation of non-existing source code.

Noticeable research. In general, there are two noticeable lines of research on utilizing existing data to assist software development.

First, many software researchers tried to mine existing data accumulated in previous software development to acquire useful knowledge for software development. Techniques with this focus generally summarize some patterns from existing data and use these patterns as guidelines for future software development. A typical example is defect prediction [6], where various attributes are extracted and a prediction model is built. Typically, existing techniques in this line can acquire patterns with only low accuracy at the current stage. Therefore, it is only feasible to use these patterns to aid human developers, but it is infeasible to perform software automation solely based on these patterns.

Second, artificial intelligence researchers have recently started studying models (e.g., neural network models) for learning from big data in software development. Their primary concern is proper treatment of the highly structural information (e.g., source code). For example, tree-based convolutional neural networks [7] are proposed to accommodate complex structures of code. These neural networks are demonstrated to be effective for distinguishing functionalities of code snippets. Conceptually, this research line complements the previous line from the perspective of software automation because automatic code generation relies on a wide range of knowledge including both knowledge specific to the development tasks and common coding knowledge. However, the accuracy of the existing techniques in this line of research can be competitive in very few tasks.

Expecting a breakthrough. A breakthrough of software automation with the help of big data may occur in the foreseeable future. There may be two criteria that determine such a breakthrough (which may be also called data-driven software automation) has occurred. First, source code for a wide range of daily software development tasks can be automatically generated. In other words, the breakthrough techniques should be able to oversee a large portion of activities for developing software with a typical size and written in a mainstream programming language. Second, the automatically generated code should have comparable or even higher quality than human-written code. In particular, the maintainability of the automatically generated code should be high enough so that human developers can work with it comfortablly.

In fact, daily software development tasks nowadays are mainly based on mature algorithmic and architectural designs. That is, instead of inventing totally new algorithms and architectures, developers usually adopt ideas and patterns from existing successful algorithms and architectures. Furthermore, with the accumulation of various useful software libraries, there seems to be a trend that daily software development becomes less creative. Thus, daily software development tasks involve more of

building with existing libraries than inventing new code.

Approaching the breakthrough. In the following, we focus on discussing where data-driven software automation might replace human developers in the near future.

First, it is more practical and viable to apply data-driven software automation during software evolution than during initial software development. During software evolution, the history of a software system is typically a burden for human developers to evolve the system, but for data-driven software automation, the history of the system can be a valuable data source. Thus, the historical data may serve as a benefit instead of a hindrance for data-driven software automation to produce quality code to evolve the system.

Second, one particularly promising scenario in data-driven software evolution is functionality transplantation, which migrates the code implementing certain functionalities from one software system to another. Compared with data-driven software evolution in the general sense, the scenario of functionality transplantation explicitly provides developers with the code that will be transplanted. In other words, a technique for functionality transplantation starts with some existing code instead of some (partial) informal specification. Thus, the search procedure can focus on a small search space around the code at hand. Another promising scenario in data-driven software evolution might be bug fixing, where some partial specification represented as test cases is typically available. Since it is convenient to execute the test cases to check the satisfiability of the partial specification, the search procedure in this scenario can be in a much simpler form.

Third, data-driven software automation may also provide useful support for developing the initial version of a given software system. Let us consider that developers are required to build a software system or sub-system using existing software libraries. On account of the abundance of software libraries, the typical daily development requirement for developers is to compose application programming interfaces (APIs) from various libraries with some simple logic. Thus, with a knowledge base that stores the up-to-date knowledge about all the known libraries, a technique for data-driven software automation may search in only a limited search space that covers common composition logics to fulfill most daily development tasks in this scenario. Our early work called the architecture-based component composition (ABC) approach [8] is an existing attempt on reuse-based software automation. Given some specifications, the ABC approach matches the specifications against existing components and finds the most suitable components. Then, the ABC approach tries to generate glue code to compose the components. Whenever the generation of glue code fails, the ABC approach allows manual generation of glue code. Typically, the glue code is tightly mapped to an underlying mechanism (e.g., a middleware system). Compared with data-driven software automation, ABC does not rely on big data to find suitable components or to generate glue code.

Limit. To conclude, we briefly discuss the limit of data-driven software automation. In our opinion, software development may not become fully automatic solely due to data-driven software automation. What has been discussed so far is to intrinsically reuse existing software rather than invent new software. The main difference between data-driven software automation and traditional software reuse is that what is reused in data-driven software automation is primarily the implicit knowledge buried in code. Reuse of existing knowledge may become a limiting factor for data-driven software automation to be applicable in the scenario of developing software without suitable precedent knowledge. Of course, this scenario may occur with low frequency in a typical software development process.

Acknowledgements This work was supported by National Key Research and Development Program of China (Grant No. 2017YFB1001803).

References

1 Xu J, Chen D, Lv J, et al. Software Automation (in Chinese). Beijing: Tsinghua University Press, 1994
2 Pressman R. Software Engineering: a Practitioner's Approach. Boston: McGraw Hill Press, 2010
3 Hudak P. Conception, evolution, and application of functional programming languages. ACM Comput Surv, 1989, 21: 359–411
4 Mernik M, Heering J, Sloane A M. When and how to develop domain-specific languages. ACM Comput Surv, 2005, 37: 316–344
5 Hutchins W, Somers H. An Introduction to Machine Translation. London: Academic Press, 1992
6 D'Ambros M, Lanza M, Robbes R. Evaluating defect prediction approaches: a benchmark and an extensive comparison. Empir Softw Eng, 2012, 17: 531–577
7 Mou L L, Li G, Zhang L, et al. Convolutional neural networks over tree structures for programming language processing. In: Proceedings of the 30th AAAI Conference on Artificial Intelligence (AAAI-16), Phoenix, 2016. 1287–1293
8 Mei H, Chang J C, Yang F Q. Software component composition based on ADL and middleware. Sci China Ser F-Inf Sci, 2001, 44: 136–151