Guest Editors' Introduction to the Special Section on Software Language Engineering

Jean-Marie Favre, Dragan Gašević, Ralf Lämmel, and Andreas Winter

HUMANKIND is defined by Language; Civilization is defined by Writing [5]. Since the Stone Age and then the shift from Prehistory to History, nothing has really changed in this respect. What remains constant is that no information can be shared without languages. But, as we are entering the so-called Information Age, we are definitively facing new challenges in Information Technologies (IT). This special section considers one of these challenges, namely, the disciplined and systematic Engineering of what is coined *Software Languages* (SL). Starting to address this challenge now may make a difference in the *Software Century* [8].

Before entering into the technical aspects of Software Language Engineering and illustrating some of them via the technical papers selected in this special section, it is worth explaining the rationale that led to the combination of the three words "Software," "Language," and "Engineering." In fact, we believe that *Software Language Engineering* (SLE) could constitute an emerging subfield of Software Engineering. We bet that its importance will significantly increase in the future as the need for new Software Languages will become even more important. The current increase in attention to Domain Specific Languages (DSL) (which is largely reflected in the papers in this special section) might constitute one trend toward this direction [2], [4], [6], [7]. We should move from the black art of language design to an engineering discipline. However, in order to put the papers included in this special section in a broader perspective, let us consider for a moment the past. Let us go back to the early ages of mankind and consider languages through the ages, and make a very quick trip from Natural Languages and the Stone Age, to Software Languages and the Information Age.

The Stone Age and Natural Languages. When anthropologists say that language defines mankind, the very term *language* obviously refers to *natural languages*. This is the

kind of languages that all humans use daily without even noticing it.

Language is not the only distinctive features of our species: The creation and use of sophisticated tools also makes homo-sapiens unique among the animal kingdom. Starting in the Stone Age, the first artificial tools were introduced for food production. Though stone technology was extremely crude, it constituted the root of craft, and craft later leads to engineering. Nevertheless, though *tools* and *languages* define mankind, they long remain two separated elements in our cultural heritage.

From Prehistory to History. The clear-cut separation between technology and language disappear with the progressive emergence of the first *IT*. The invention of *writing* [1] is indeed one of the most important successes in the history of IT; it marks the move from Prehistory to History. Whatever the technology actually used and the epoch considered, IT has always been about techniques and tools for *symbol management*. In this context, the notion of *language* refers to the *shared conventions* that allow production of valid symbol combinations and interpretation of those symbol combinations. This corresponds to what we know today as the syntax and semantics of languages, but, beyond this distinction, a much more fundamental and basic question is raised: Since these conventions are so important, how are they to be shared?

Though *explicit language descriptions* is one of the characteristics of software language engineering, history clearly shows that, in practice, languages do *not* have to be represented explicitly and precisely to be successful. The invention of writing played a fundamental role in the emergence of first civilizations, but its huge success did not rely on any formal description of the syntax or semantic. How can this be? While the transmission of natural language is considered to be a natural process, this is not the case for written languages. Among all civilizations, the lack of explicit language descriptions has been compensated for by the creation of adequate social structures ensuring that *conventions* could be shared over space and time. In antiquity, language knowledge was transmitted by confirmed scribes to their apprentices in the context of "home of knowledge" and this mostly by means of examples and through oral tradition [5]. Note that the learning process of new languages largely remains the same, even in the context of informatics: Students rarely learn programming languages through the examination of the language grammar and formal semantics. Explicit language descriptions could help in some circumstances, but as long as

- J.-M. Favre is with One Tree Technologies, Luxembourg, and the University of Grenoble, France. E-mail: jean-marie.favre@imag.fr.
- D. Gašević is with the School of Computing and Information Systems, Athabasca University, 1 University Drive, Athabasca, AB T9S 3A3, Canada. E-mail: dgasevic@acm.org.
- R. Lämmel is with the Institute for Computer Science, University of Koblenz-Landau, Universitätsstraße 1, Koblenz, Germany. E-mail: laemmel@uni-koblenz.de.
- A. Winter is with the Department of Software Engineering, Carl von Ossietzky University Oldenburg, Uhlhornsweg 84, Oldenburg, Germany. E-mail: winter@informatik.uni-oldenburg.de.

For information on obtaining reprints of this article, please send e-mail to: tse@computer.org.

human activity can compensate the lack of them, they are not strictly required, at least in the context described so far.

Early Computers and Machine Languages. The invention of writing definitively shaped the last five millennium; let us now move directly to what constitutes one of the next biggest steps of the history of IT, namely, the invention of effective computing devices such as punch cards machines, followed in the middle of the 20th century by the emergence of early computers. Until then, human brains were the only information processing units on Earth. It then became clear that both computers and human brains could process symbolic information. *Computers* could be seen as *symbol management machines*. They are able to “read,” “write,” and move around large amounts of symbols and all of this is done at an unprecedented speed. Obviously, the machine’s operator had to program the machine in order to get the result. To do so, the operator had to first “write” another set of symbols that could be interpreted in terms of the “machine language.” The term machine language was really appropriate at that time: Linguistically, the operator was totally constrained by the set of operations provided by the machine and communicated with it by using a machine language. In some sense, early operators were at the service of the machine, not the other way around.

Beyond Computer Science. Times are changing. Nowadays, the balance has turned back to the human factor. When compared with early computers such as the ENIAC, current symbolic machines are rather inexpensive. They are just considered another device available for the sake of humankind. In the software world, “Computer Science” retained the attention of researchers for a long time. It should be clear, however, that nowadays Software Engineering *must* be considered in a more comprehensive scope; in the context of Informatics: the discipline that considers both computer *and* humans. After all, software users, and software engineers are people too and software engineering is a human and social activity. As a matter of fact, let us consider the evolution of software engineering actors. In the second half of the 20th century, “coders” were progressively replaced by “programmers” and then by large teams of “software engineers” with different skills and roles. In this special section, particular attention is being paid to some rather new roles, namely, “domain experts” and “software language engineers.” Informatics is no longer restricted to the sole activity of programming.

Beyond Computer Programs. “Software” and “program” are no longer synonyms. Modern software is obviously made of programs, but it also includes all sorts of symbolic information, such as requirements, specifications, scenarios, models of all kind, user documentation, test cases, bug report, database schemas, software repositories, software quality documents, software configurations, deployment descriptors, tutorials, and licensing schemes, to name only a few. What is more, as large open source projects and complex software ecosystems testify, modern software production has turned into a highly collaborative and social activity. As a result, software engineers make growing use of forums, wikis, and RSS feeds. These constitute other examples of software artifacts. Software is more than computer programs.

Beyond Computer Languages. All of these software artifacts are expressed using some languages, whether formal or not, whether explicitly defined or not, whether well-structured or not. As a result, there are really plenty of languages in the software landscape. Just consider for a moment the following unordered list of terms that one might hear in the context of conferences about software: *rule-based languages, formal specification languages, configuration languages, meta-languages, query languages, model-transformation languages, schema definition languages, requirement specification languages, domain-specific languages, protocol-definition languages, scripting languages, text formatting languages, business-process description languages, architectural description languages, markup languages, modeling languages*, etc. If you spent some time in thinking about it, you could fill another page with such terms. These terms attempt to identify categories of languages according to different features, but this is not the point. What is important for the sake of this discussion is that programming languages constitute only a small part of these languages. So, what should we collectively call this very broad range of languages?

The term “Computer Languages” might be considered as a potential candidate. As a matter of fact, it is often used as a generic term for this purpose. However, we believe that using that term supports a profound misunderstanding in our profession. The term Computer Language strongly suggests that languages are designed solely for computers, i.e., machines. This view is really unbalanced. Who can argue seriously that computers care about the successive improvements characterized through the history of programming languages? For instance, was structured programming invented for the sake of computers? Or was it instead to solve human cognitive limitations in the presence of goto statements? Did we create higher level programming languages created to communicate with computers, only being able to understand machine language? Were modular languages invented for computers or to facilitate human activity in the context of collaborative development? In fact, the terms computer languages and machine languages do not convey the importance of human aspects in their design.

Software Languages. In order to refer collectively to all kinds of languages somehow involved in software production, we prefer instead to use the term *Software Language (SL)*, hence the combination of the first two words of this special section title. What is fundamental to grasp here is that *each software language should indeed be considered as a single coin but with two sides*: one side for the machine and the other for the software engineers. Both sides have to be designed very carefully in order for the language to be successful. On the machine’s side, we find so-called “computation models” (e.g., state-machine or lambda-calculus if we consider abstract machines). That is where Computer Science fits. But, since software engineers are people, not machines, we should not forget the human’s side of software languages. This second side is about perception, cognition, and usability, that is, the human brain is the processing unit of symbols that need to be grasped. In fact, when considering the history of humankind, the new aspect with software language is that this is the

first time that human brains have to communicate with other processing units, also known as the symbolic machines, which are able to work with symbols as well.

Software Language Engineering. Now that the term "software language" has been defined and this choice justified, we should explain why considering software language from an *engineering perspective* is so important. After all, natural languages have proven extremely powerful, and nothing has been done for this to work! The development and evolution of first IT technologies such as writing has required substantially more effort. Still, the evolution of writing systems has been very slow and when it has not been, due to hazardous events, it has been the result of the work of generations of craftsmen. No engineering has been present there. What about the development of programming languages then? Doubtlessly, the situation is different in this case. There are plenty of tools for generating parsers from grammar specifications, for instance. In this context, scientific knowledge about formal languages is applied in the context of practical needs. So, parser construction can be considered as an engineering activity. However, Software Language Engineering is much more than defining languages, building parsers, and compilers.

As software languages (artifacts) are software too, all of the software engineering concepts could also be applied to the development, maintenance, and evolution of software languages. One should consider, for instance, *software language lifecycle, requirements analysis for software language, software language quality assurance, risk analysis for SLE, reverse engineering of software languages, software language metrics, software language documentation, software language testing and validation, software language architecture, software language versioning, software language refactoring, software language optimization, software language analysis, software language evolution, co-evolution of languages and languages-dependent artifacts, SLE economics and cost estimation, methodologies for software language engineering*, etc.

Interestingly enough, many of the SLE topics cited above are actually addressed somehow in the papers selected for this special section. The reader will certainly agree with the fact that, currently, most SLE activities are either implicit or are performed in an ad hoc way. Consider, for instance, textual or graphical syntax design, which is fundamental when a software language is considered from the human's side. Did cognitive experts design the syntaxes of popular languages such as UML? The answer is definitively no.

Software Language Engineering in the context of the Information Age. Even though SLE seems to be a valid research field, one may argue that software languages such as C++, HTML, or PHP have been built in an empirical way they are still very popular and useful. Nevertheless, except for a few languages that are somehow successful, why is it that many languages simply die just after they were born? How many languages (properly designed or not) are in the cemetery of software languages? And, when a language such as C++, Java, or HTML becomes so widely adopted, what is the cost of evolving it, maintaining all of its successive versions, and fighting for all of its deficiencies? Just like for software, the cost associated with the provision

and evolution of a new language is not so much about its initial development, but also about its deployment in operational contexts, its maintenance and evolution, etc. Even clear characteristics of estimating the quality of these languages are missing.

One valid argument to move from Software Language Craft to Software Language Engineering is therefore to optimize the chances of success of SLE projects and to optimize the corresponding costs and qualities. Though this argument is certainly valid, at least to some extent, the success of ad hoc yet popular software languages could raise some doubts. Nevertheless, there is another argument that makes us believe that SLE may receive more attention in the future. As new social organization schemes for software production are likely to appear, there may be a sustainable need for new software languages. Remember the many kinds of languages that the term software language conveys. Again, programming languages are just particular examples. The need for new languages is likely to continue because of an increasing number of specialties in software engineering. What is more, the Information Age is expected to be characterized by an unprecedented amount of heterogeneous information. This century will see the emergence of so-called *ultra-large scale systems* or systems of systems. Experts from many different domains will have to collaborate in order to build such systems. At the same time, they will need tools to represent the knowledge in their own fields. *Ubiquitous computing* (also known as ubiquitous symbol management) seems to be the future of IT [3]. This may well mean *ubiquitous software languages*.

Beyond the Cast of Computer Scribe. It is unlikely that ultra-large scale systems could be built if all knowledge is to pass through the hands of a very small cast of scribes keeping for themselves the secret of symbols. The 20th-century IT scribes, aka coders, have to understand that, just as writing skills have spread all over our societies, this is a very good thing. In fact, learning to read and write is one thing, but what is more important is how to apply these basic skill in particular professional fields. For instance, lawyers "write" just like account managers, mathematicians, and chemical engineers, but what they write in the context of their profession is "domain specific." They all refer to different concepts and techniques. Some of them use special-purpose languages [1]. The time of scribes concentrating all areas of knowledge is definitively over. This could be the same in the context of software. Today's computers are in everybody's hands, but built-in software with a predefined set of questions-answers may be enough for regular users, but not for future business, scientific, and engineering purposes. Relying on coders is just like relying on street scribes. Instead of this situation, software language engineers should give languages to professionals so that they can run their own business with symbolic machines suited for their needs. It is more than unlikely that the Information Age will rely on a small cast of scribes concentrating all knowledge. The need for automatic symbol management will be pervasive in the future, so there is a good case for Software Language Engineering and ubiquitous languages.

Software Linguistics. This raises our last question in this editorial: If engineering is an application of scientific

knowledge to solve practical (industrial) cases, what is the scientific field which corresponds to Software Language Engineering? More generally, what is the science that studies languages? The answer is obviously Linguistics! Since software languages are languages too, *we envision the development of Software Linguistics as an emerging scientific discipline* as a complement of *Natural Linguistics* and under the umbrella of (general) Linguistics. Programming languages and natural languages are very often opposed, but we believe that the general framework of linguistics can be applied to software languages as well. While this special section focuses on Software Language Engineering and its technical aspects, studying software languages from a broader perspective could only be beneficial.

About this special section. The aim of this special section is to gather together a wide range of novel research work in the area of Software Language Engineering. In response to the call for papers, we received 29 submissions. Each paper was reviewed by at least three expert referees. After the two rounds of the peer-review process, we selected six papers that, all together, represent the variety of topics and challenges in Software Languages Engineering.

The first paper in this section introduces some important software language matters. In “A Flexible Infrastructure for Multilevel Language Engineering,” Colin Atkinson, Matthias Gutheil, and Bastian Kennel put special emphasis on a language aspect that is too often neglected or misunderstood: the fact that a combination of symbols can be both considered from a linguistic and ontological perspective. Based on these two dimensions, the authors elaborated a conceptual framework for multilevel language engineering, and describe an SLE environment that shows how both dimensions could be taken into account at the same time.

The papers that follow are arranged (at least to some extent) according to the position where they put the cursor in the duality between the syntax of languages and the concepts that these language enable us to express. The first three papers concentrate mostly on the syntax of languages. In contrast, the last one, which is almost syntax-free, deals with the challenge of building a unifying language by combining concepts from other languages. Let us start with the syntax side.

In “The ‘Physics’ of Notations: Toward a Scientific Basis for Constructing Visual Notations in Software Engineering,” Daniel L. Moody argues against the myth that “syntactic sugar” is an unimportant aspect of languages. He argues that the choice of symbols is, on the contrary, really important from a cognitive point of view. The physical appearance of a language indeed plays a fundamental role in language adoption, during communication among the various software stakeholders, and, in fact, during all human intensive activities based on the language. This paper considers the human’s side of syntax and argues that concrete syntax design is an important SLE issue.

The next paper also deals with syntax, but from the machine’s side, and considers a reverse engineering perspective. In “Grammar Recovery from Parse Trees and Metrics-Guided Grammar Refactoring” by Nicholas A. Kraft, Edward B. Duffy, and Brian A. Malloy, the authors address the situations where languages already exist and are instrumented by a compiler or parser, but where no

explicit language description is available. Again, this is a very common situation in practice and, as pointed out before, languages can be used (by humans) without an explicit and formal language description. This constitutes an issue, however, when new language processors have to be derived in a partial or fully automated way. Kraft and his colleagues show how the grammar of a language can be recovered from parse trees. This paper also shows that, just like for any other software artifact, computing metrics and applying refactoring also make sense in the context of Software Language Engineering.

The next paper, “Engineering of Framework-Specific Modeling Languages” by Michal Antkiewicz, Krzysztof Czarnecki, and Matthew Stephan, is not only about creating explicit language descriptions, but also about creating explicit languages. This paper exhibits yet another kind of situation where SLE is of great importance. It deals with what can be called *implicit software languages*, *virtual software language*, or *proto software languages*. In contrast to the software languages mentioned so far, proto software languages do not (yet?) have the full status of explicit software languages: 1) They do *not* have any syntax, neither implicit nor explicit; neither abstract nor concrete, and 2) the concepts related to the languages are usually just present in software engineers heads. The importance of shared conventions was mentioned earlier, but in the context of proto (software) language (just like in the context of proto-writing [1]), conventions are not fixed in stone. They rely instead on a social agreement of what constitute “good practices.” Frameworks and API documentation constitute common examples of what constitute proto software languages. Antkiewicz and his colleagues show how such virtual languages can be materialized in the form of so-called *framework-specific modeling languages*. This paper also provides a nice example of a carefully defined SLE methodology.

In “A Model-Based Approach to Families of Embedded Domain Specific Languages” by Jesús Sánchez Cuadrado and Jesús García Molina, a rather similar problem is tackled, but, instead of “recovering” a modeling language, their approach is based on the notion of *embedded languages*. The idea is to define a new language on top of an existing one (referred to as the *host language*). Languages with invasive syntax do not provide many facilities for implicit language extension. In contrast, syntax-light languages such as Lisp or Haskell provide much better opportunities to define what could look like a domain specific syntax. In their paper, Sánchez Cuadrado and García Molina show how a model-driven SLE approach could be used to define in a systematic way new DSLs on top of the Ruby language. They also address the problem of the modularity of languages and show how DSLs can be composed. This paper indeed illustrates the need to consider *software language architecture* as an important aspect of SLE.

Finally, the list of papers selected ends with “FAML: A Generic Metamodel for MAS Development” by Ghassan Beydoun, Graham Low, Brian Henderson-Sellers, Haralambos Mouratidis, Jorge J. Gomez-Sanz, Juan Pavón, and Cesar Gonzalez-Perez. The problem addressed in this paper is related to the proliferation of multiple software languages within the same domain of knowledge. As a matter of fact, in all domains, different teams all over the planet act

somehow as providers of (implicit or explicit) languages. While *language competition* is a rather natural process when a new field is explored, successful engineering process at the international level should lead sooner or later to some language standardization efforts. This raises both social and technical issues. Beydoun and his colleagues show how metamodels can be used for *language (re)conciliation*.

It has been a pleasure for us to work on producing this special section. In fact, it is the result of a substantial effort from the emerging SLE community: We would like to thank the many authors who submitted manuscripts for their dedication in writing papers in line with the topic of this special section; we would also like to thank the reviewers for their wonderful job in carefully reviewing all of these papers in time, commenting on them, and providing useful insights.

Languages are wonderful objects of studies and, as shown in this introduction, Software Language Engineering has many different facets. While it is natural to focus on particular facets to solve technical problems, we should never forget that one of the core challenges is also to put all of these facets together to form the big picture. Software Language Engineers should be prepared to deal with ubiquitous software languages as they are likely to play a role in the Software Century [8].

We hope that you will enjoy reading this compilation as much as we did assembling it.

Jean-Marie Favre
Dragan Gašević
Ralf Lämmel
Andreas Winter
Guest Editors

For more information on Software Languages, Software Language Engineering, and Software Linguistics, please visit <http://planet-sl.org>.

REFERENCES

- [1] F. Cajori, *A History of Mathematical Notations*. Open-Court Publishing, 1929.
- [2] S. Cook, G. Jones, S. Kent, and A.C. Wills, *Domain-Specific Development with Visual Studio DSL Tools*. Microsoft.NET Development Series, Addison-Wesley Professional, 2007.
- [3] A. Greenfield, *Everyware: The Dawning Age of Ubiquitous Computing*. New Riders Publishing, Mar. 2006.
- [4] R.C. Gronback, *Eclipse Modeling Project: A Domain Specific Language (DSL) Toolkit*. Addison-Wesley Professional, 2009.
- [5] *The First Writing, Script Invention as History and Process*, S. Houston, ed. Cambridge Press, 2008.
- [6] S. Kelly and J.P. Tolvanen, *Domain-Specific Modeling: Enabling Full Code Generation*, 2008.
- [7] A. Kleepe, *Software Language Engineering—Creating Domain Specific Languages Using Metamodels*. Addison-Wesley Professional, 2008.
- [8] B. Boehm, "Making a Difference in the Software Century," *Computer*, vol. 41, no. 3, pp. 32-38, Mar. 2008.

Jean-Marie Favre received the MSc and PhD degrees in informatics from the University of Grenoble, France. He is a software anthropologist and a software language archeologist at the University of Grenoble, where he also serves as an assistant professor. He is a visiting scientist at One Tree Technologies, Luxembourg, and a member of the Laboratory of Informatics at Grenoble (LIG). He has published approximately 90 papers and coedited a book (in French)

Beyond MDA: Model Driven Engineering. He has been invited as a keynote speaker and/or to give tutorials at more than 10 international events and summer schools. He has organized various national and international events and he is deeply involved in community engineering concerning Research 2.0, Software Engineering 2.0, and XFOR 1.0.

Dragan Gašević received the Dipl.Ing., MSc, and PhD degrees in computer science from the University of Belgrade in 2000, 2002, and 2004, respectively. He is a Canada Research Chair in Semantic Technologies and an assistant professor in the School of Computing and Information Systems at Athabasca University and an adjunct professor in the School of Interactive Arts and Technology at Simon Fraser University. He is a recipient of Alberta Ingenuity's 2008 New Faculty Award. In his current research activities, he investigates relations of semantic technologies with software language engineering, technology-enhanced learning, and service-oriented architectures. He has (co)authored more than 200 research papers and is the lead author of the book entitled *Model Driven Engineering and Ontology Development*. While serving as an associate editor and editorial board member of six international journals, he has also edited special issues in journals such as *IET Software*, *SoSym*, the *IEEE Transactions on Software Engineering*, and *Information Systems*. While serving on the steering committee of the International Conference on Software Language Engineering (SLE), he has also been a keynote speaker, organizer, chair, and member of program committees of many international conferences.

He is a recipient of Alberta Ingenuity's 2008 New Faculty Award. In his current research activities, he investigates relations of semantic technologies with software language engineering, technology-enhanced learning, and service-oriented architectures. He has (co)authored more than 200 research papers and is the lead author of the book entitled *Model Driven Engineering and Ontology Development*. While serving as an associate editor and editorial board member of six international journals, he has also edited special issues in journals such as *IET Software*, *SoSym*, the *IEEE Transactions on Software Engineering*, and *Information Systems*. While serving on the steering committee of the International Conference on Software Language Engineering (SLE), he has also been a keynote speaker, organizer, chair, and member of program committees of many international conferences.

Ralf Lämmel holds the Dr.-Ing. degree from the University of Rostock, Germany, and is a professor of computer science at the University of Koblenz-Landau, Germany. He has held positions at CWI (the Dutch Centre of Computer Science and Mathematics), the Free University of Amsterdam, and Microsoft in Redmond, Washington. His main research focus is on grammar-based software language engineering techniques. He is one of the founders of the International Summer School Series GTTSE—Generative and Transformational Techniques in Software Engineering.

Andreas Winter holds the chair of Software Engineering at Carl von Ossietzky University Oldenburg. Prior to his appointment in Oldenburg, he was affiliated with Johannes Gutenberg University Mainz, the University of Waterloo, and the University of Koblenz, where he received the PhD degree on meta-modeling in 2000. Current research includes model-driven software development, service-oriented tool interoperability, software migration and software-(re)engineering processes. He served as program chair of the 12th and 13th European Conference on Software Maintenance and Reengineering, and is member of the steering committees of the International Conference on Software Language Engineering and the European Conference on Software Maintenance and Reengineering.