Like any other scientific field, computing emerged as a brilliant idea implemented by brilliant scientists. Unfortunately, the only users of computers for many years were the brilliant scientists who invented them-because only they knew the language that the machines understood. Computers would never have revolutionized the world if the user base of those who could talk to computers had not expanded. Over the past six decades, computer "communicators" have evolved from brilliant scientists computing shell trajectories to grandparents dialing up to check their e-mail. More importantly, computer *languages* have evolved to meet the demands of the computer users. The continuation of this evolution is the joining of design tools and the executable system, in the form of Model-Integrated Computing (MIC).

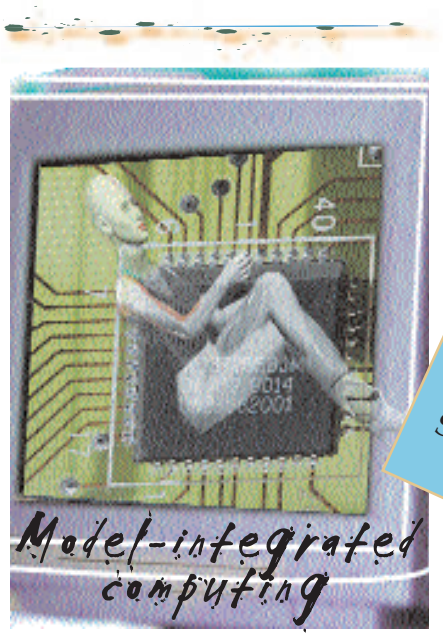
The integration of design tools and an executable system is an important step. For the past 10 years, software engineering has been something of an embarrassment to the computer science field. The primary reason is its methodologies seldom work for all domains. For example, the emergence of the internet brought into focus the need for a methodology to design software not as a centralized application but as a client/server-which requires a new approach when building that software. This results in a new practice or methodology every few months to make up for the domain on the next horizon (e.g., autonomous agents or grid computing).

Computing and domains

Computers have evolved significantly since Turing's codebreaking machine and ENIAC in the 1940s. Computer programming has also evolved since then: from interconnecting wires to punch cards, all the way to hand-written assembly code and later high-level languages.

The first high-level languages created (FORTRAN and COBOL) were instituted because of their relevance to particular domains (FORTRAN for engineering and COBOL for business calculations). Before FORTRAN and COBOL, programs were written either with punch cards or in assembly code. Thus, to be a coder in those days knowledge of the specific computer hardware was required.

Creating the code was difficult, as the coder (the one who wrote the program) and the program designer (i.e., the one who wanted the program) were seldom the same person. Therefore, it was necessary for the program designer to explain the program's goals and objectives to the coder. This process was time consuming and error prone. Also, once written, the code was not trivial to analyze as it was written on such a low level. FORTRAN and COBOL allowed the gap between the coder and program designer to shrink, as well as provided more readable code.



Jonathan Sprinkle

A *domain* is a family of related systems. For example, the domain of engineering uses numerical analysis, matrix operations, and optimization solvers to achieve its goals. FORTRAN, as one of the first high-level textual languages, was designed specifically for use with the engineering domain, and in its prime was considered necessary knowledge for an engineering programmer. C/C++ and Java have replaced FORTRAN as the "must know" language for employment today. Unfortunately, whereas FORTRAN was more domain-specific than assembly code (i.e., it was targeted specifically toward the engineering world), C/C++ and Java are less domain-specific for engineering than FORTRAN was. Language preferences aside, although C/C++ and Java may not be the best programming language for engineering, they *are* tools that provide a knowledgeable programmer with the ability to create domain-specific environments in the form of the application.

Domain-specific environments

A domain-specific environment provides a domain expert with tools and an interface to use a computer to help solve problems in that domain. Software for engineering computer-based systems has proven to be useful to design engineers and programmers. MATLAB (heavily based on FORTRAN), with its interpretive environment, allows for quick evaluation of written code without compilation. This ability provides engineers with an interface for complex mathematical manipulations in conjunction with frequently used engineering transforms (Bode plots, transformations into the frequency domain, etc.). Computer applications such as SPICE (circuit specification through a text file) have emerged for circuit analysis. Experts in domains have learned to harness the computer to aid them in solving problems.

In addition to system and software designers, end users have also benefited from domain-specific environments. The Microsoft Office Suite is an application is designed for use in the office (spreadsheet, presentations, e-mail, document preparation). The Mathematica tool is designed specifically for solving mathematical problems by directly inputting mathematical equations. Autodesk's AutoCAD is a computer aided drafting tool designed for creating technical drawings and blueprints from computer models.

Each application, and others too numerous to mention, was created using a programming language such as C/C++ with one particular domain in mind. In each case, programmers abstracted the domain-specific problems that the applications aimed to solve into domain-independent algorithms that C/C++ was designed to solve.

Computer programmers created the integrated development environment (IDE) to aid them when creating new applications. The IDE is a special application that provides the ability to create, edit, debug, and generate new applications. By creating an application for the development of applications, programmers were able to "bootstrap" the application development process.

Domain-specific modeling environments

Later, thinking on a higher, or "meta" level, computing researchers realized that the creation of domain-specific environments was *itself* a domain. The IDE (as an application for creating

applications) was a first step toward this realization, but the IDE was still too open-ended (that is, not every application created by an IDE could function as a design environment for a domain). If it were possible to create a domain-specific environment designed specifically for creating domain-specific environments, then the overhead required for developing small-scale environments (e.g., creating a software class structure and basic programming overheads), and the difficulties of maintaining large-scale environments (e.g., documentation management, coding conventions) might be alleviated. Using this domain-specific IDE, every application that was generated would be a design environment customized to a particular domain. This is a restricted view of the purpose of programming, and is most suitable for domains with well-defined abstractions that are used to compose a design's behavior. The use of these abstractions as programming elements is referred to as *modeling*.

In the engineering world, a *model* is a mathematical abstraction that explains and/or predicts the behavior of a physical artifact. Model-integrated computing (MIC) is the technique of using models to describe an application or computer-based system. Model-integrated program synthesis (MIPS) uses MIC to produce the model, and then from that model produces the computer program that is the executable code (also known as the executable model) of the computer-based system.

Users that employ the MIC or MIPS approach have been successful users of domain-specific modeling environments (DSMEs). This is because a DSME is a convenient way to customize and work with a finite set of interacting "models" that describe a computer-based system. Before utilizing the MIC or MIPS approach, a trade-off analysis is performed. The analysis amortizes the amount of labor required to create the appropriate domain-specific environment, vs. the amount of labor saved in creating a solution using a domain-independent environment. For the right systems, MIC can be extremely beneficial (e.g., gate-level specification using four different gates), but could also be more trouble than it is worth (e.g., building a graphical language for C++). DSMEs are different from IDEs in one major way: IDEs are used to specify *software*, and DSMEs are used to specify *systems*. This allows the implementation to be decoupled from the specification, and is the major advancement of the DSME over the IDE as the design environment for complex systems.

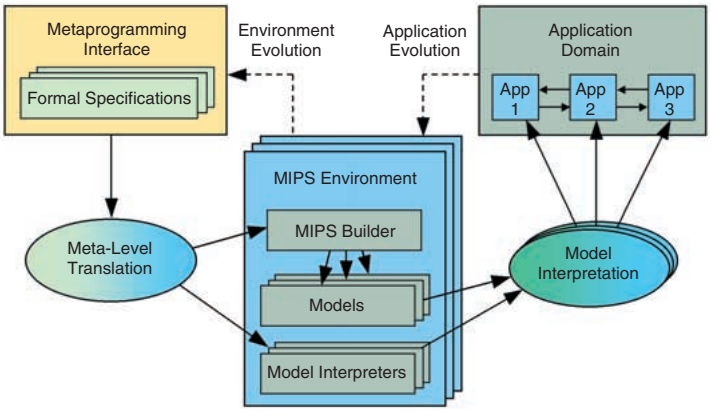


Fig. 1 Overview of Model Integrated Program Synthesis (MIPS)

The metamodel

The method for the rapid creation of domain-specific environments is to create a metamodeling environment whose purpose is to create domain-specific environments. In a domain (such as CAD or accounting) only certain portions of the physical world have an impact. For example, it is not necessary for an accountant to know how much L2 cache her processor has when computing her taxes. The essence of the domain-specific environment is that *only* those things important in the domain are available to the domain user. The metamodeling environment then should be able to create a representation for anything that might exist in a domain. Once all actors of the domain have been represented in the metamodeling environment, then the domain-specific modeling environment can be created. A DSME is a domain-specific environment that uses models and/or MIPS to create systems. Because such a great deal of meaning is incorporated into the elements of the DSME, a significant amount of the low-level details can be extracted from the domain elements themselves, without requiring the modeler to worry about them. Returning again to a gate-level design example, a NAND gate has an easily recognized symbol, and a well-understood behavior. Using a special type (with a picture to represent the gate) a DSME can attach behavioral semantics to that type, and use those semantics when generating the final output of the system. This becomes more worthwhile when complicated objects (such as counters or multiplexers) are available for use in the DSME.

The metamodeling environment makes rapid development of these DSMEs possible. Today's software practices can take years of man-hours to produce prototypes of a system, while metamodeling and MIC have been shown to take appropriate systems from start to finish in a matter of man-months (a DSME designed for the dynamic reconfiguration of Saturn's Delaware plant required two man-weeks of programming and two hours for installation). The biggest advantage of the DSME is that it advances what COBOL and FORTRAN started: making computing accessible to more domain experts, and allowing for the reduction of menial labor required in producing and maintaining a computer-based system.

Evolution with MIC

MIC and MIPS truly shine in the area of evolution. One of the fundamental ideas of MIC is that the size of the change in specification should be proportional to the size of the change of

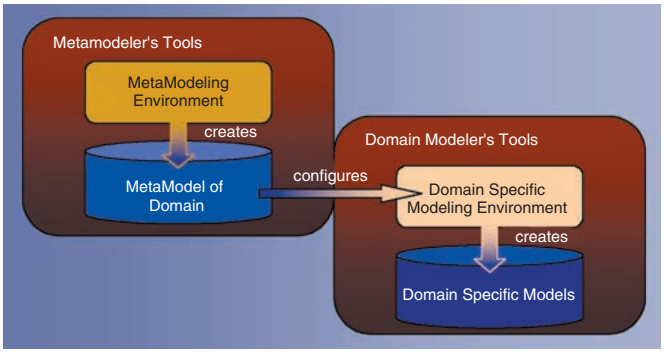


Fig. 2 The cascading design approach. Metamodeling tools are used to design a domain specific modeling environment. This customized environment is then used to develop the models of the system.

the implementation. Changing the sampling time of a system, for example, should not require a change in the software architecture. MIC has several layers of specification that are subject to change, so it is important to understand the context of each type of evolution. Figure 1 is an overview of the types of evolution from the perspective of the computer-based system, and Fig. 2 details the cascading approach of developing the tools first, and using the custom generated (i.e., domain-specific) tools to develop the system models.

Application evolution

Application evolution is the evolution of the computer-based system execution models (see Fig. 1). The applications that drive some computer-based systems are implemented with no plans for upgrade because developers think they will be replaced (e.g., the Y2k problem). However, others are implemented with specific guidelines for maintenance, which may or may not be followed. Still others are implemented with no thought put into the question at all.

Traditional software and system engineering has depended on software experts to maintain engineering systems. This means that even small changes to an already implemented computer-based system requires a software expert to understand the changes, and then investigate the ramifications of those changes to the already implemented code. The change process can result in unintended consequences; e.g., system downtime. In general, the change process is not a rapid one as the software expert is not always familiar with the engineering portion of the system.

Since with MIC the behavior and structure of the system is specified using models, then any changes to that behavior are made to the models. Then the updated system is generated from the models and installed. The right side of Fig. 1 gives an overview of the MIPS evolution cycles. Application evolution is facilitated by the MIPS environment, which is developed for the domain. Notice that in order to modify the execution models, changes are made to the domain models (i.e., the above layer).

Environment evolution

Another kind of system evolution is the evolution of the MIPS environment itself. Computer-based systems that are not well designed are difficult to maintain. Also, the poor analysis of the requirements of a computer-based system can result in delayed deployment due to unsuccessful operation or inadequate performance during field-testing. The computer-based systems with the best designs have usually been engineered with good design principles and rigorous analysis of the mathematical models that govern the system. Good designs take into account all of the factors of the system and examine all of the entities of the domain to determine the solution.

However, no matter how rigorous the analysis or strict the adherence to good design principles, computer-based systems can, and do, eventually change in their definition. That is, the factors and entities present in the original design may have changed or been removed. The resulting system may require maintenance/updating not only on the instance level, but also on the *metamodel* level. This is known as environment evolution. Environment evolution is required when any formal specification of the domain is changed. If a component of a system is no longer available for installation (for example, modeling amplifiers using transistors rather than tubes) then the DSME should no longer provide a tube as a possible part, but should instead give the proper transistor. After changing these formal specifications, the meta-level translator is used to generate the new MIPS environment. As was true in application evolution, to affect changes to this layer of modeling, the layer above is changed-in this case, the metamodel.

Conclusion

The integration of design tools and an executable system is an important step in software engineering's evolution. Model Integrated Computing through the use of Domain Specific Modeling Environments is an emerging approach to computer programming. By providing a customized level of abstraction in a relatively short period of time, and leveraging existing domain knowledge of by creating the language specifically for a domain expert, DSMEs are a logical progression of system design technology. MIC is the technology that turns a design tool into an executable system.

DSMEs should be used only when they fit the profile required by the domain. A domain with a manageable set of components with well-understood behaviors is an excellent candidate for a DSME, as the final computer system can be generated from the model of the system. Once the domain is identified, then it is possible to use metamodeling to develop a language that suits that domain. To quote Mark Twain, "The difference between the right word and the almost right word is the difference between lightning and the lightning bug." MIC is the practice of finding the perfect words to express the problems of a domain, and using the implied meaning of the language to rapidly and efficiently implement the system.

Read more about it

* Lohr, Steve. *The Programmers who Created the Software Revolution - Go To*, Basic Books, 2001.

* MetaCase Consulting, "MetaEdit+ Revolutionized The Way Nokia Develops Mobile Phone Software," MetaCase Technical Report, Jyväskylä, Finland, 2002.

* E. Long, A. Misra, J. Sztipanovits, "Increasing Productivity at Saturn," *IEEE Computer Magazine*, vol. 31, no. 8, Aug 1998, pp. 35-43.

* J. Garrett, A. Ledeczi, F. DeCaria: "Towards a Paradigm for Activity Modeling," *2000 IEEE International Conference on Systems, Man, and Cybernetics*, Nashville, TN, October, 2000.

* J. Sztipanovits, G. Karsai, "Model-Integrated Computing," *IEEE Computer Magazine*, vol. 30, no. 4, Apr 1997, pp. 110-112.

* J. Miller, et al., "What UML Should Be," Communications of the ACM, vol. 45, no. 11, Nov 2002, pp. 67-85.

About the author

Jonathan Sprinkle is a postdoctoral researcher at the University of California, Berkeley. He received his Ph.D. in 2003 from Vanderbilt University in Nashville, TN, where he performed research in metamodel based environment evolution. He received his Masters from Vanderbilt in 2000, and his bachelors i*n cursu honorum* with a double major in Electrical Engineering and Computer Engineering from Tennessee Technological University in Cookeville, TN, where he served as student body president in 1998-99.

During his graduate career Jonathan was the recipient of an IBM Fellowship, held the position of Master Teaching Fellow in the Vanderbilt University Center for Teaching, and was selected as one of 10 young researchers in the United States to attend the 52nd Meeting of the Nobel Laureates in 2002.