Automatic Code Generation Through Model-Driven Design

Patrick W. Burke
Philip Sweany
University of North Texas, Denton, Texas
October 23, 2007

Abstract **–** Software engineering has not been able to automate software development to a degree necessary to take full advantage of the hardware features available in current computing technology. This lag in capabilities can be attributed, in a large part, to the lack of sufficiently complex and capable automation tools within the software community. This paper investigates the feasibility of using automatic code generation as a direct output of current model-driven design systems as a possible solution to this noted deficiency. A number of different approaches to modeling and code generation are discussed along with the near-term and future possibilities for each approach. The discussion results in a conclusion that the level of technical capability of the current systems is insufficient to provide a true solution to the general problems facing software engineering, but further research and development of the modeling-generation systems could very well encompass the solution at some time in the future.

*Key Words:*Model-driven development, code generations, software specification, code proofs, autocode process

I. INTRODUCTION

The field of software engineering has had a checkered past. The same discipline which enabled human travel to the moon and back, pin-point 3-dimensional global positioning, and advanced manufacturing robots has also caused the loss of multi-billion dollar space vehicles[1], total failure of communication networks, and the compromise of supposedly "secure" personal identification data.[2] Obviously, the software errors that led to these catastrophic system failures were not intentionally included in the delivered systems. They were, instead, insidiously introduced as a consequence of incomplete or incorrect system specifications or testing. Thus, the challenge to the software engineering industry is now, as it has been for the past 40+ years, to determine how these software mistakes, affectionately referred to as "bugs" within the software community, can be kept from ever being introduced in the first place. As a minimum, the mistakes must be found early in the requirements or design phases to avoid the financial disaster and loss of credibility associated with finding the problems after delivery. According to Dr. Barry Boehm, one of the icons of Computer Science and Software Engineering, "Finding and fixing a software problem after delivery is often 100 times more expensive than finding and fixing it during the requirements and design phase."[3] Even worse, software development is responsible for more than 80% of program/design delays and, unfortunately, over 33% of software developments produce a final product which is not within 50% of its predesign expectations.[4] Clearly, something must be done to reverse this trend. Automatic code generation (ACG) is one possible contributor to the solution, and will serve as the focal point of this paper. ACG cannot, in and of itself, however, correct the software engineering problems faced by the industry today. Code cannot be miraculously generated out of primordial ooze from whence we came. It needs some kind of human-generated model or design as its basis. One plausible predecessor of automatically generated code, and the solution proposed by this paper, is to utilize ACG in concert with Model-Driven Development [or Design] (MDD) as the front end of the software development process.

Section II provides an historical timeline of software development paradigms to provide the background necessary to describe the current trends in software development. Section III presents a synopsis of current MDD practices, their perceived short-comings, and research opportunities within the field. Section IV describes the current state of ACG along with its challenges and areas which will require further research.. Section V ties everything together with the conclusion of this paper, namely, that the use of MDD in conjunction with simulation and automatic code generation can, in time, contribute significantly to a reduction in design development and implementation costs with a beneficial side effect of increasing the reliability of the product.[5]

II. SOFTWARE DEVELOPMENT PARADIGM HISTORY

In the beginning, there was chaos. Software development of the 1950's and 60's was best typified by a group of "coders," who locked themselves in the "computer room" and performed computer magic. The software process, at the time, was totally hero driven. The computers of the era were behemoths, consuming massive space and copious amounts of energy. The hardware was maintenance intensive and of limited capability and reliability. Programming was essentially a maximization problem of extracting maximum computation from the limited computer capacity. All of this began to change when electronic hardware began to make exponential processing improvements with the introduction of a single revolutionary innovation ... the integrated circuit. Miniaturization and production innovations, such as CAD systems, have allowed computer hardware to continue to achieve exponential growth, creating obsolescence issues within 3-5 years of release of continually evolving new products. In the early 1980's, before the advent of the internet to the masses, electrical engineers had the Transistor-to-Transistor Logic (TTL) Catalog of electronic components which described all the components available to build bigger and better systems

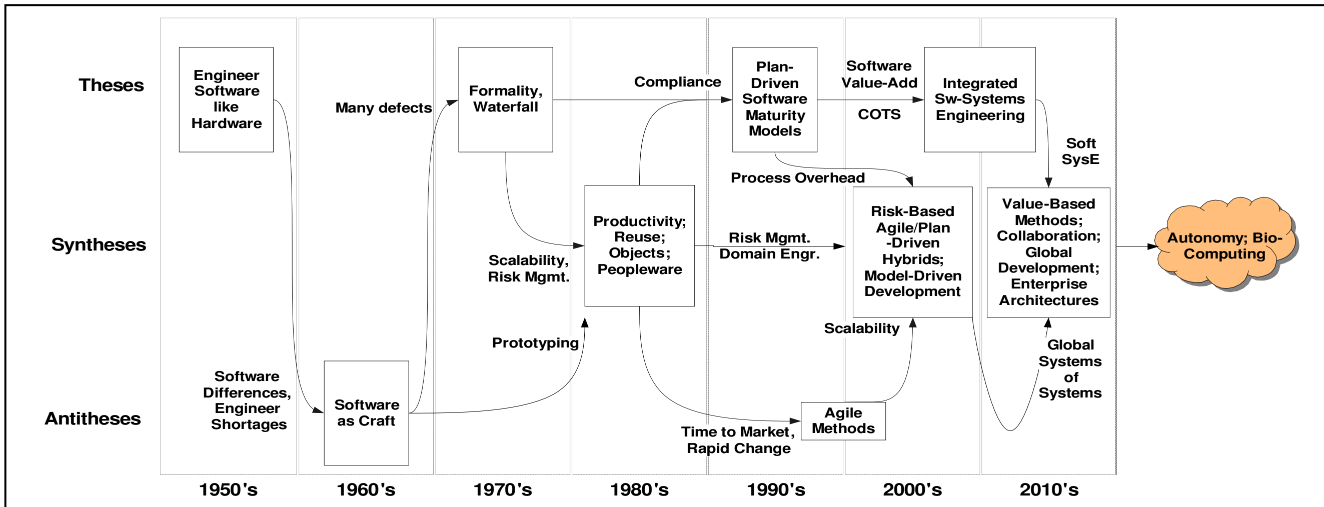


Figure 1. Software Engineering Evolution (by Dr. Barry Boehm)

along with their pin outs, or external interfaces. When software engineers can hold up a book of software components like that, they will be able to keep up with the hardware engineering progress. Software engineering is still looking for that "Holy Grail".

This quest has traveled many roads, each with its own promises and problems. Dr. Boehm provided a concise history of paradigms and a vision for future innovations[6], as show in Figure 1. His historical perspective clearly gives credit to some old friends (e.g. crafting and waterfall), whose utility has yielded to the more contemporary developments of Maturity Models, Risk Management, and Agile Methods. All of these current methods are highly dependent upon large-scale processes which are highly people-dependent and greatly reliant upon manual reviews to ensure correctness and completeness of the product. The quality of the constituent reviews is directly proportional to the skill level and availability of the key reviewers. The only automation associated with these processes is the electronic capture, initiated with manual input, of the artifacts for the benefit of the reviewers. Orders of magnitude level improvements in software engineering are unachievable in such an expert-dependent system. Introducing significant automation tools into the software engineering toolset is the key to solving the problem.

That having been said, it is important to acknowledge one major difference between software engineering and the bulk of the other engineering disciplines: The end product of software engineering is not a physical object. It is, instead, a binary coded representation of human thought. As such, software development will always be, to some degree, dependent upon smart, innovative, and skilled human beings[7]. Tools developed by this elite group, however, will allow exponential growth of error-free, quality software systems created by software engineers not as highly skilled in the domain of software engineering as in the domain of the application for which the developed system is being created. Artisans will always be needed in all fields of software engineering.

Current software development methods employ processes and certifications in an attempt to reduce the volatility in the developmental execution. Most notably, the Software Engineering Institute (SEI) based at the Carnegie Mellon University and funded by the United States Department of Defense (DoD) has developed a succession of corporate software engineering certifications to encourage the documentation and sharing of best practices within a given organization and, on rare occasions, with the rest of the DoD contracting community. The original certification system was simply named SEI Levels. Rules were spelled out dictating the criteria for each level and the amount of time that had to be served at one level before being allowed to apply for certification at the next level. The SEI Level system was later replaced by the Capability Maturity Model (CMM) for software engineering and later by the Capability Maturity Model Integrated (CMMI) for entire engineering organizations. These evaluation systems utilize organizational normalization of development metrics using statistical control methods in conjunction with specifically tailored plans and procedures to capture current execution measurements which can be employed in the continuous improvement of the organizational plans and procedures. The statistical control methods, such as 6-Sigma, had their roots in the control of factory processes.

Risk Management has also become a major player in any current large software development program. Risks are evaluated and quantified in terms of schedule and dollars from the proposal stage through initial delivery. The most important part of each risk assessment is the risk mitigation plan which is created to proactively reduce the chances of the risk becoming reality. Each numbered risk and its associate mitigation plan is re-evaluated on a periodic basis, at least monthly, until the risk is realized or mitigated to a sufficiently low probability of realization.

Design description has developed independent of the development methodologies. All program description paradigms and products have essentially one goal: describe the desired system in a human-friendly form providing abstraction from the details of the software design and hardware implementation. Flow charts were used with the earliest software systems, but quickly became cumbersome and inadequate as software systems grew in size and complexity. Pseudocode promised more versatility and capability for design descriptions and is still in limited use today. It suffered, however, from the use of the ambiguous English language and was found to be cumbersome to use in describing interfaces between software pieces. By far the most successful and the most universally accepted design paradigm came with the concurrent introduction of Object-Oriented Design (OOD) and Unified Modeling Language (UML). Most modern software systems are developed using these tools.[8]

Unfortunately, the plans, procedures, reviews, and design systems which constitute the basis of the current software developmental methodologies only produce an ancillary affect on the actual creation of the executable developmental code. In general, the same actual coding methods which were in vogue in the 1970's are still being employed today.[9] The languages have changed and the design description systems have changed, but the conversion from design to code still relies on the skills of the individual coder.

Dr. Boehm's vision of the future includes descriptive names of the software engineering tools of the future. Model-Driven Development, Integrated Software-Systems Engineering, Systems of Systems, Enterprise Models, Autonomy, and Bio-Computing all include viable fields of research within the field of software engineering. MDD leading to ACG is one tool which is well within the ability of current technologies to implement. "UML, simulation, and code generation contributes significantly to a reduction in design, development, and implementation costs."[10] Yet, generating code is of questionable value unless the code can be absolutely validated, that is, proven to be correct.[11] Furthermore, the coding phase of the current software engineering methods is relatively short. If manual reviews of the automatically generated code are required, this technology represents a very limited improvement, if any, over the current methodologies.

III. MODEL-DRIVEN DESIGN/DEVELOPMENT

Model-Driven Design is a software development paradigm which uses predominantly non-textual visual representations to produce an abstracted view of a software system. It is the embodiment of the old adage "A picture is worth a thousand words." The acronym MDD actually has two decodings in literature: Model-Driven Design" and "Model-Driven Development." While the two decodings are used almost interchangeably, "Development" implies a complete cycle of software development. "Design," on the other hand, tends to imply utility mainly in the design phase. Since there appears to be no significant distinction between the two terms, either definition will suffice within the context of this paper.

"MDD provides the ability to go from diagrams (including State Charts) to source code (automatic code generation). The technology has been providing major benefits of shorter design cycles and high quality products to developers using OO."[12] The mantra of current software development processes is "faster, better, and cheaper," so MDD is a good candidate solution to the historic software development problems.

There are several requirements for any MDD system to be of maximum utility. In general, MDD requires design concepts and supporting model languages that are abstract enough to construct designs in which no specific platform constraints are imposed. At the same time, such concepts should be expressive enough to allow the construction of designs at a sufficiently detailed level to describe how the design can be realized."[13] The user interface should be simple, easily understood, and unambiguous. To avoid being locked into any particular vendor product, the MDD tool should ascribe to some universally accepted, internationally standardized modeling languages, such as Simple DirectMedia Layer (SDL) or UML.[14] UML with its state charts, activity charts, sequence charts, and collaboration charts, is by far the most widely industry-accepted language used to model the abstract behaviors of target designs.[15]

UML, in its current instantiation, however, provides insufficient features, tools, and constructs to fulfill the MDD to ACG dream. Many consider UML to be burdensome and complicated in its detailed implementation. As complicated as UML may be, though, it still lacks the ability to associate semantic meaning to a model at the depth required for full automatic code generation. Before this can be achieved, a significant amount of research will have to be conducted in each of the following areas, at a minimum:[16]

- Viewpoint conflict analysis
- Feature interaction analysis
- Formal specification techniques
- Formalization of semantics in modeling languages
- Reuse of development experience (software reuse)
- Patterns
- Domain-specific modeling languages
- Systematic software testing
- Compilation technologies for models

While not a requirement, per se, verification of a model must be strictly validated. One major purpose of a system model is for use as a system prototype to be used in the requirements phase to find and eliminate errors early when their correction is least expensive."[17] To be effective as a prototype, the MDD tool must allow direct execution of the model, allowing relatively easy validation of the model with given sets of input data and/or sequences and their associated expected results. Creation of an executable model via a

standalone software development task is unacceptable as it adds increased probability for introducing error in the implementation phase, independent of the actual design. Executable UML was mentioned as a feature of the Rhapsody (Telelogic), Rational Rose (IBM), Requirements to Design to Code (NASA), and BridgePoint UML Suite (Mentor Graphics) development systems in some of the referenced literature. Executable UML and easy, if not automated, verification of the model as truly representative of the requirements of the design of the target system allows for quick validation of the model at any point in the development.[18] Validation of the system after each correction or update is the ideal process to be followed, albeit predictably expensive. Incremental validation would prevent the introduction of multi-layered, interdependent errors. "Constant validation of the system is the key to producing high quality systems very quickly."[19]

An almost unavoidable reality of real-time software systems is the need to incorporate hand-written, highly specialized, and extremely efficient code segments in order to meet tight timing requirements.[20] It is even possible that legacy code, code from a third party, or hardware-dependent code may have to be included in the design model to make it truly representative of the nuances, processing, and timing of the target systems. For all these reasons, an MDD system which is expected to accurately model a target system must allow incorporation of external code segments into the design and/or generated code. It must be understood, however, that the more code that can be generated automatically from a verifiably correct model, the less likely that the human developers will introduce errors.[21] Furthermore, a reverse-engineering capability is a highly desirable feature of any MDD system. Such reverse-engineering supports manual correction/modification of models or code and allows the system to create the complimentary piece. This feature must also allow incorporation of external code or model updates to be automatically incorporated.[22]

The requirement to fully validate a model cannot be overstated. The use of MDD as a prototype requires the model to mimic a representative usage profile of the target application to flesh out requirements and architectural issues early (i.e. before the project proceeds into the design and implementation phases).[23] Such models can be regarded as executable specifications.[24] One method of verification, input-output sets, has already been discussed, but that method can be tragically flawed due to resource and schedule limitations which necessarily prohibit the creation and execution of an exhaustive set of tests. One novel solution to this limitation is called a formal proof of correctness. It has been proposed using a patent-pending system developed by NASA Goddard Flight Center called R2D2C (Requirements-to-Design-to-Code):

"In the R2D2C approach, engineers (or others) may write requirements as scenarios in constrained (domain-specific) natural language, or in a range of other notations (including UML use cases). These will be used to derive a formal model that is guaranteed to be equivalent to the requirements stated at the outset, and which will subsequently be used as a basis for code generation. The formal model can be expressed using a variant of formal notations. Currently, we are using CSP, Hoare's language of Communication Sequential Processes, which is suitable for various types of analysis and investigation, and as the basis for fully formal implementations as well as automated test case generation, etc."[25]

Obviously, the rigors of any formal proof of software will require some sort of formal specification[26] and the weakest link in this allegedly verifiable model is the reality that "while engineers are happy to write descriptions as natural language scenarios, or even using semiformal notations such as UML, they are loath to undertake formal specifications."[27] Another drawback of proof of correctness methodology is the cost. To be useful, the proof must be valid from requirements specification through code generation. This approach requires the generation of independently provable inferences for the theorem prover and the accumulation of knowledge rules to implement laws of concurrency.[28] These are not tasks which can be assigned to inexperienced engineering staff members. As such, the proof of correctness methodology may be practical for only small projects or projects which implement safety-critical requirements in which the loss of human life or extremely important/expensive mechanical systems are at risk. Applying the proof of correctness to a safety-critical thread within a larger software system may be an interesting area of future research. In fact, Dr. James Vallino suggested the topic of using modeling in conjunction with theorem provers to mathematically prove safe operating scenarios within safety-critical systems, but did not indicate that any follow-up research was ever conducted.[29]

The concept of MDD and even automatic code generation is not new. It has been a topic of interest in the computer industry for decades. Certainly, the lack of adequate modeling tools has been a primary issue, but the software development norms and processes have also hindered the acceptance of an automated coding process. Since its inception, software engineering has relied upon a system of stepwise refinement in the development of large software systems. The historic requirements to create a high level design leading to a detailed design leading to code provided formal mechanisms for the understanding of the software product at many levels of abstraction. Managers and customers could understand the high level design without the complication of the low level details. Coders could interpret the mid-level details of the detailed design in order to produce the code, and so on. The fear in the software engineering community has always been that MDD, especially MDD leading directly to automatic code generation, will bypass the historically proven paradigm of stepwise refinement with its associated formal reviews. In its current instantiation, however, MDD provides a semblance of stepwise refinement with different model views. While the actual names and partitions differ somewhat among MDD systems, all resemble the standard modeling views

described in the Object Management Group's (OMG) Model Driven Architecture Guide:[30]

- Computation Independent model (CIM)
  - Describes the requirements for a system and the business context in which the system will be used. The model describes how a system will be used, not how it is implemented. CIMs are often expressed in business or domain-specific language.
- Platform Independent model (PIM)
  - Describes how the system will be constructed, without reference to the technologies used to implement the model. It does not describe the mechanisms used to build the solution for a specific platform.
- Platform Specific model (PSM)
  - Describes a solution from a particular platform perspective. It includes the details that describe how the CIM can be implemented and how the implementation is realized on a specific platform.

The OMG MDA was formalized in 2002-03, culminating in the release of the *MDA Guide*. Since then, the model-driven concept has continued to evolve, resulting in the current notion of Model-driven Engineering (MDE). Unlike MDA, MDE is not constrained in the number or types of models that can be constructed to describe a system. The tendency toward more complex software systems is the motivation driving research in the area of MDE. The focus of MDE is to use multiple levels of abstraction from a variety of perspectives to completely describe and specify the desired software system, the code for which will subsequently be automatically generated. MDE also promises automated support for transforming and analyzing models. In such a development system, the models themselves, rather than the code, will be the primary artifacts of software development.[31]

Exactly how these models fit in the historic formal review process is a current topic of heated debate within the software engineering community, but the fact that the models, in the order shown, provide the desired element of stepwise refinement is paramount to their utility. In addition, UML provides mechanisms for hiding and expanding details of a model.

As these new technologies are being developed, the software engineering processes are being modified to incorporate the new tools and products. The historical design review is being replaced with a "Model Review," of some sort, which has the following minimal goals:

- check whether or not the textual specified functional requirements are realized in the model
- ensure that relevant modeling guidelines are fulfilled (e.g. naming conventions, structuring, modularization)
- check that a number of selected quality criteria such as portability, maintainability, testability are met
- check that the implementation model meets the requirements for the generation of safe code (e.g. robustness) and efficient code (e.g. resource optimizations)"[32]

This review is a critical link in the validation of the model as representative of the functional requirements of the system. Unfortunately, the effectiveness of this review is highly dependent upon the expertise of the reviewers. While this may be seen as a weakness in the process, human verification of the functionality is a necessary evil at some point in the process. As mentioned earlier, artisans will always be needed in software engineering.

IV. AUTOMATIC CODE GENERATION

Once a design or model is accepted as representative of the functionality of the desired system, it must be converted to an executable module. Traditionally, the design, as described by graphic and/or textual representations, is manually converted into source code which is then compiled, linked, and built as a separate deliverable product. Of course, the code product itself undergoes unit level, functional level, and system level testing. It is subjected to code reviews, integration testing, test readiness reviews, and formal acceptance testing. More often than not, the testing, which empirically encompasses the longest development phase, results in changes to the requirements, design, or interfaces ... and the whole process is repeated. For decades, researchers have studied ways to short-circuit this repetitive cycle to produce better and cheaper software faster. Given the current trend toward the use of executable UML to produce verifiably correct models as the front end to a direct compiler, the dream of automatic code generation may be one step closer. "Automatic code generation results in higher productivity and uniformly high quality. Modeling helps in early detection of errors in the application development cycle."[33] Furthermore, some researchers claim generation of 80-90% of the final application code directly from the validated model as a typical benchmark.[34]

Many of the requirements and desired features discussed with relation to MDD systems also apply to the automatic code generation (ACG) systems. The ACG system must be user-friendly, allow the direct insertion of external code, and convert (reverse engineer) that code back into the model.

Ideally, at the end of the design or modeling phase, the user would simply press the ACG button and the development could be declared complete. Realists, defined as anyone who has ever endured the rigors of a software development cycle, understand, however, that testing is the hallmark of the development cycle. Unless a code generation system can be shown to be provably correct, some degree of testing must be employed. Since current debuggers run at the code level[35], it is imperative that the automatically generated code be human-readable, completely self-documented, and unambiguous. All programmers strive for this exact clarity of their code with varying degrees of success. How much

more difficult will this goal be to attain within the constraints of an automatically generated set of coded instructions? This problem will need significant amount of research and development in the coming years in order to create viable ACG systems to the satisfaction of the general software development community.

There are other potential problems which could possibly cancel the benefits of automatically generated code. When modeling has indicated a problem in a system, the cycle-time required to affect a small change and verify its correctness has been found to be significantly greater for the model-based approach than the traditional approach.[36] If the number of problems found is determined to be very low, the cost-benefit analysis could still favor MDD-ACC, but if a significant number of changes are needed, the cost of using MDD-ACG could be prohibitively high. Another problem with ACG is that the code generated for use in testing on a generic processing platform, which is the normal mode of operation for concurrent software-hardware product development projects, must then be regenerated for a target processor. If all testing must be repeated on the target hardware, the utility of the intermediated code must be seriously questioned.[37] Also, since ACG is a fairly new concept, in practical terms, it has not been studied in large, complex development projects. Much work needs to be done and documented with relation to the scalability of the MDD-ACG solution.

Another possible topic for future study is the application of formal proofs of correctness for automatically generated code. While this has been the goal of NASA's R2D2C system, such a rigorous proof has proven to be very expensive. Establishing the proof requires statement-by-statement analysis of the program, a process that involves the creation of numerous mathematical statements called verification conditions (VD's). The proof of the VCs requires significant skill, especially when the specification has abstracted away significant amounts of details."[38] So, a formal proof of correctness can practically be used in only very limited circumstances when the cost can be justified.

In the meantime, software engineering will have to rely on more traditional methods of verification: code reviews and testing. Autocode reviews differ from traditional code reviews in that autocode reviews consider both the model and the code simultaneously. Of course, the model is completely validated before the code is generated, so the autocode review must verify that the generated code truly represents the model. Until code generators (CGs) are trusted enough to allow their use without reviewing the generated code, autocode reviews are still a good software engineering practice."[39] The goals of the autocode reviews follow:

- find errors that have been introduced by inappropriate modeling or faulty use (e.g. configuration) of the code generator
- identify problems or errors in the model or in the autocode that are difficult to detect in the model but easier to find in the autocode
- ensure that custom code parts are properly integrated
- identify possible efficiency improvement, since they are easier to detect on code level to reveal code generator errors"[40]

The autocode review, like its counterpart the model review, will require the participation of a very talented and diverse group of systems and software professionals to provide the skill level necessary to make the review beneficial. Time and experience can be expected to eventually increase the reliability of the automated systems and, if warranted, slowly erode the need for such labor-intensive reviews.

Relying on testing to prove the functionality of a system is, at best, a hit and miss venture. It can be considerably difficult in complex software systems where it is infeasible to conduct the number of test cases required to establish a high level of confidence in the system. It can be fairly effective, however, if use cases are used to formally specify the functionality of the system. More requirements being explicitly described in the form of unambiguous use cases leads to systems more completely verified by testing. As noted earlier, the testing phase has historically far exceeded its allotted scheduled duration due to incomplete or incorrect requirement specifications or testing. All errors, whether they be requirement, design, model, code, or test case, must be fixed when found in the test phase. A perfect code generation system always produces incorrect code when the system requirements are wrong!

V. CONCLUSIONS

Automatic code generation via model-driven design is an emerging technology that is showing great promise as a candidate for making software development faster, better, and cheaper. Sophisticated modeling tools that can correctly represent the specification and functionality of software systems are currently in use and under development. To date, application of these tools has been extremely limited due to the complexity inherent in large, multiprocessing systems. "However, just as mechanical devices can unleash creative potential by amplifying physical effort, programming tools can improve software development by helping developers manage details, find inconsistencies, and ensure uniform quality."[41]

The field of MDD/ACG is replete with opportunities for further research which will allow large-scale acceptance of MDD/ACG within the software engineering community. MDD/ACG provides an environment within which component-based software reuse may finally become reality. This will require significant research in the areas of formal specification techniques, model compilation techniques, and the formalization of semantics in modeling languages. Additional research will be needed with regard to the processes which will guide software development in the MDD/ACG arena, as well as, in the areas of software automation tools and test capabilities to support such development. Finally, much research is required to validate

the performance of the models and automatically generated code, either through formal proof of correctness or in some less formal type of software testing. Given the cost of formal proofs, research might even be conducted in limited application of formal proofs within a safety critical thread of a much larger software system. Ultimately, the only factor which will hold the key to the success or failure of any software technology is its impact on the bottom line of companies and institutions using that technology. While the current state of MDD/ACG may certainly be far from perfect, it is stable enough to warrant wide-spread use for current software development efforts. It is only through such use that the hidden problems can be identified and corrected to produce an optimized solution to the problems faced by the software engineering community. In time, MDD-ACG technology will make a significant improvement in software development efficiency and make the artisans of the craft more productive, appreciated, and indispensable.

VI. REFERENCES

[1] "Mars Climate Orbiter Team Finds Likely Cause of Loss," NASA Press Release 99-113, 30 September 1999 <http://mars.jpl.nasa.gov/,s[98/news/mco990930.html>.

[2] "Some Recent Software Failures Caused by Software Bugs", SoftwareEngineeringReference.com, <http://www.sereferences.com/software-failure-list.php>.

[3] B.W. Boehm and V.R. Basili, "Software Defect Reduction Top 10 List," Computer, Jan 2001: 135.

[4] Jerome L. Krasner, "UML for C Developers," Embedded Market Forecasters, April 2005 <www.embeddedforecast.com>: 5.

[5] Krasner 7.

[6] Barry Boehm, "A View of 20th and 21st Century Software Engineering," Proceeding of the 28th International Conference on Software Engineering ICSE'06, (Shanghai, China 25 May 2006): 16.

[7] James R. Larus, Thomas Ball, Manuvir Das, Robert DeLine, Manuel Fahndrich, Jon Pincus, Sriram K. Rajamani, and Ramanathan Venkatapathy, "Righting Software," IEEE Software May 2004: 92.

[8] Krasner 5.

[9] Larus, Ball, Das, DeLine, Fahndrich, Pincu, Rajamani, and Venkatatathy 92.

[10 Krasner 7.

[11 Michael G. Hinchey, James L. Rash, adn Christopher A. Rouff, "A Formal Approach to Requirements-Based Programming" Proceedings of the 12th International Conference and Workshops on the Engineering of Computer-Based Systems, 4-7 Apr 2005: 343.

[12 Krasner 6.

[13 João Paul Almeida, Remco Dijkman, Luis Ferreira Pire. Dick Quarel, Marten van Sinderen, "Abstract Interactions and Interaction Refinement in Model-Driven Design," Proceedings of the 2005 Ninth IEEE International EDOC Enterprise Computing Conference, 19-23 Sep 2005: 273.

[14] Yuefent Zhang, "Test-Driven Modeling for Model-Driven Development," IEEE Software Sept/October 2004: 285.

[15] Chen Xi, Lu JianHua, Zhou ZuCheng, and Shang YaoHui, "Modeling SystemC Design in UML and Automatic Code Generation," ASP-DAC'05: 933.

[16] France and Rumpe, pp. 41-43.

[17] Krasner 10.

[18] Krasner 12.

[19] Krasner 10.

[20] Krasner 8.

[21] Hinchey, Rash, and Rouff, *A Formal Approach to Requirements-Based Programming:* 5.

[22] Krasner 13.

[23] Vinay Kullkarni and Sreedhar Reddy, "Generating Enterprise Application from Models -- Experience and Best Practices (2005), Advances in Object-Oriented Information Systems: OOIS 2002 Workshops, 2002, <http://www.softmetaware.com/oopsla2005/kulkarni.pdf>: 274.

[24] Ingo Sturmer, Mirko Conrad, Ines Fey, and Heiko Dorr, "Experiences with Model and Autocode Reviews in Model-based Software Development," SEAS'06 (ACM: Shanghai, China, 23 May 2006): 46.

[25] Michael G.Hinchey, James L. Rash, and Christopher A. Rouff, "Some Verification Issues at NASA Goddard Space Flight Center," <http://vstte.ethz.ch/Files/~hinchey-rash-rouff.pdf>: 5.

[26] Elizabeth A. Strunk, Xiang Yin, and John C. Knight, "Echo: A Practical Approach to Formal Verification," FMICS'05 Lisbon, Portugal 5-6 September 2005): 44.

[27] Hinchey, Rash, and Rouff, "A Formal Approach to Requirements-Based Programming": 2.

[28] Hinchey, Rash, and Rouff, "A Formal Approach to Requirements-Based Programming": 4.

[29] James Vallino, "Software as a Component in Safety-Critical Systems Professional Development Leave Report," <http://www.se.rit.edu/~jrv/publications/Vallino-SoftwareSafety-LeaseReport.pdf> (2005): 4.

[30] Joaquin Miller and Jishnu Mukerji, eds., *MDA Guide Version 1.0.1*, (OMG: 2003), http://www.omg.org/docs/omg/03-06-01.pdf, pp. 2-5 thru 2-6.

[31] Robert France and Bernhard Rumpe, "Model-driven Development of Complex Software: A Research Roadmap," Future of Software Engineering (FOSE'07), Minneapolis, MN 23-25 May 2007: 37.

[32] Sturmer, Conrad, Fey, and Dorr 46.

[33] Vinay Kullkarni and Sreedhar Reddy: 2.

[34] Krasner 10.

[35] Krasner 8.

[36] Vinay Kullkarni and Sreedhar Reddy 7

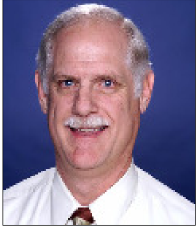
[37] Yuefent Zhang 84

[38] Strunk, Yin, and Knight 44.

[39] Sturmer, Conrad, Fey, and Dorr 47.

[40] Sturmer, Conrad, Fey, and Dorr 47.

[41] James R. Larus, Ball, Das, DeLine, Fahndrich, Pincu, Rajamani, and Venkatatathy 92.



Pat Burke has been an IEEE member since 2001 and was a member of the first class of IEEE Computer Society Certified Software Development Professionals in 2002. He has worked as a software engineer/manager for Raytheon since 1984. He holds a BSCS from the Air Force Academy (1977), a MSCS from the University of North Texas (1984), and is currently enrolled as a Computer Science and Engineering PhD student at UNT. He is married with two children and a grandson. He is also an FAA Certified Flight Instructor.

Phillip Sweany, associate professor in UNTs Computer Science and Engineering Department, maintains a research focus in both compiled optimization for architectures exhibiting fine-grained parallelism and application of compiler optimization algorithms to authomated synthesis of net-centric systems.