01 Jan 2006

Practical Experiences in using Model-Driven Engineering to Develop Trustworthy Computing Systems

Thomas Weigert
Missouri University of Science and Technology

Frank Weil

Recommended Citation

T. Weigert and F. Weil, "Practical Experiences in using Model-Driven Engineering to Develop Trustworthy Computing Systems," *Proceedings of the IEEE International Conference on Sensor Networks, Ubiquitous, and Trustworthy Computing (SUTC'06) (2006, Taichung)*, Institute of Electrical and Electronics Engineers (IEEE), Jan 2006.
The definitive version is available at https://doi.org/10.1109/SUTC.2006.1636178

Practical Experiences in Using Model-Driven Engineering to Develop Trustworthy Computing Systems

Thomas Weigert and Frank Weil
Motorola
Schaumburg, Illinois, USA
{thomas.weigert, frank.weil}@motorola.com

Abstract

In this paper, we describe how Motorola has deployed model-driven engineering in product development, in particular for the development of trustworthy and highly reliable telecommunications systems, and outline the benefits obtained. Model-driven engineering has dramatically increased both the quality and the reliability of software developed in our organization, as well as the productivity of our software engineers. Our experience demonstrates that model-driven engineering significantly improves the development process for trustworthy computing systems.

1. Introduction

Motorola has more than 15 years of history deploying model-driven engineering techniques to develop highly reliable network elements for large-scale telecommunication systems. Model-driven engineering has dramatically increased the quality and reliability of the developed software as well as the productivity of the software engineers.

This paper describes the model-driven engineering approach Motorola has been deploying. Model-driven engineering relies on capturing an application design in domain-specific languages (in Motorola, specifications are expressed using UML profiles). The specifications are formally verified for consistency (that is, the system behavior is deterministic) and completeness (at each system state that is not a terminal state further system behavior is defined). The specifications are also subject to validation by operationally interpreting the specification and through executing formally defined test cases (written at the level of the design model in a test-specific notation, say TTCN) against this specification. Domain-specific programming knowledge is captured in code generators that transform these high-level designs into optimized product software targeted to the chosen platform.

We will summarize the productivity and quality gains observed and estimate how widely these techniques are applicable to the development of telecommunications applications. We then relate this general discussion to trustworthy computing systems. This information should enable the readers to decide if model-driven engineering will be beneficial to their respective organizations.

2. Conventional Software Development

The conventional software development process begins with capturing product requirements in design models, characterized by informal diagrams and pseudo-code. Even if modern specification languages, such as UML, are being used to capture the designs, these languages are typically used to develop informal diagrams with unclear or imprecise semantics. These diagrams are then hand-translated by a team of software developers into product code in the target language. The hand-written code undergoes inspection and testing and is finally deployed at the target.

A workflow following these lines (often packaged in process frameworks such as waterfall, spiral, incremental, or rapid prototyping) is still the norm in most software development organizations. Unfortunately, this workflow is subject to several problems that contribute to the oftentainted reputation of software engineering.

Firstly, the informality and imprecision of the notations used to capture product designs tends to lead to misunderstandings between developers, in particular, when development is conducted in a globally distributed manner. More often than not, it is more luck than planning when the design diagrams are interpreted consistently across geographically dispersed organizations. What one usually observes instead is that when separately developed components are assembled into the final product, they do not work fully together. In particular, misunderstandings due to the informality of the designs in error situations or exception scenarios lead to the introduction of defects that cause product failures.

Proceedings of the IEEE International Conference on Sensor Networks, Ubiquitous, and Trustworthy Computing (SUTC'06)
0-7695-2553-9/06 $20.00 © 2006 IEEE

Secondly, the translation of design documents into code by hand is error-prone and slow. The resultant artifacts are difficult to reuse in similar applications since much of the implementation detail of the product is intertwined with the code derived from the designs.

Finally, defects are repaired at the level of the hand-written code resulting in design documents that become hopelessly out of synch with the code and become incomplete or, worse, misleading. If testing reveals serious misunderstandings of requirements, it may be more efficient to abandon the outdated design model and the current version of the hand code than it is to try to patch the hand code in place, even given the accompanying loss of productivity.

3. Model-Driven Engineering

Model-driven engineering proposes a software development process that starting from product requirements aims to capture designs in standardized high-level notations with well-defined semantics. A precisely defined semantics of the design model allows verification techniques to be applied to the model. For example, state-space exploration can reveal concurrency pathologies or other hard-to-find defects in the design. If, in addition, one is able to operationally interpret the product designs, the correctness of these designs can be established through simulation. Ideally, test cases are derived from the requirements and the designs are verified against these test cases. The designs are then translated into product code by a code generator. Finally, the resultant code is subjected to tests (again derived from the requirements) and is deployed on the target platform.

The model-driven engineering process does not make any assumptions regarding the life cycle model deployed, short of starting with an understanding of the requirements and resulting in the delivery of a software product.

Neither does model-driven engineering presuppose any particular software design methodology. This framework works best if the notation chosen to capture a software design offers concepts that are close to its application domain as this will ease stating and verifying the software designs. This process, however, does require that the chosen software design notation has clearly defined semantics in order for the designs to have well-defined meaning which can form the starting point of code generation.

Another assumption built into the model-driven engineering process is that it is desirable that a complete design of a product can be verified and that code can be generated for the complete design, not just for header files and stubs, as is the case with most development tools in use today.

Viewed in more detail, we divide model-driven engineering into two sets of activities: In application engineering, we execute the model-driven engineering process to develop products based on their software requirements. In addition, model-driven engineering supplements application development with the development of domain-specific capabilities.

In domain engineering, we first try to find a notation that is as close to the application domain as possible. The closer the concepts of the design notation are to the concepts of the application domain, the easier it will be to capture the designs, and the more likely it will be that the designs are correct.

Secondly, we need to identify techniques to verify that the design documents are correct, that is, that they reflect the application that we intend to build. This may involve the development of simulation tools or of mathematical techniques such as model checking.

Finally, we need tools to translate the design documents into product code executing on the chosen target platforms.

Domain engineering generates a set of assets that can be drawn upon in application development. Rather than code, following a model-driven engineering process, the assets are the capabilities to produce software in a selected application domain. In application engineering, these capabilities are then deployed to produce a particular product.

3.1. Model-Driven Engineering Example

The following example shows how this development framework is leveraged in practice. It is widely used throughout Motorola and has led to large development savings.

Every communication system sends information between devices. The units of information that are communicated between devices are referred to as protocol data units (or PDU for short). A PDU is typically represented in an application-internal form that is geared towards ease of access during computation. When a PDU is sent from one device to another, it has to be transformed into a stream of bits subject to the rules of the communication protocol. Every communication device, therefore, must have a software layer that will encode a PDU into a stream of bits on the sender side, and reconstruct this PDU from the incoming stream of bits on the receiver side.

Due to the low-level nature and the bit manipulations required, this data marshaling code is very tedious and error-prone to produce. This code is subject to stringent performance constraints as it is applied to every piece of information passed between devices, but these constraints are quite different between infrastructure and subscriber devices (time criticality being the primary issue for the former, space limitations for the latter). Together with differences in computing platforms, middleware layers, and error handling strategies, the difference in performance constraints results in two separate marshaling routines typically being produced, one for each type of hardware.

To support the development of data marshaling code, when following a model-driven engineering process, the first question to be addressed is how to capture the designs for these routines.

Elements	Type	Length	
Pdu Type	1	5	
Call Id	1	14	
TX grant	1	2	
TX req permission	1	1	
Encryption control	1	1	
Speech service	1	1	
Notification Id	2	6	
TX party type Id	2	2	
TX party address SSI	2	24	C[a]
TX party extension	2	24	C[b]
Proprietary	3		R

a. Present if TX party type Id is SSI or TSI
b. Present if TX party type Id is TSI

Figure 1. Tetra PDU definition: Downlink-transmit-granted (ETS 300-392-2, Table 74)

Figure 1 shows an excerpt from the TETRA standard that shows the definition of the downlink-transmit-granted PDU. As shown, the standard describes this PDU as consisting of a number of fields, each of a given type (which determines how that field is encoded as a stream of bits) and a given length. This table is already a good description of the design because the task is to produce a program that takes the information that comprises this PDU and turns it into a stream of bits that obeys the rules laid out by this table and vice versa. Verification becomes a simple comparison between the printed tables and a machine-readable representation of these tables. What is then needed is a code generator that produces this program. The program must translate the information carried in each field of the PDU into a stream of bits, segment it into chunks matching the appropriate word size, and place it at the right place within the overall stream, potentially prefixing it with any additional encoding required by the protocol. The resultant stream of bits for a simple example PDU is shown in Figure 2 (the bits added due to the encoding rules of the protocol are cross-hatched).

We applied this idea first to the development of TETRA radio handsets. The engineering team had originally estimated that it would take 18 months to develop the marshaling code for this product family. Using the model-driven engineering approach, it took only four months to develop not only the code that was shipped in the product but also the code generators that performed this task.

Once this process is in place, the assets are the language used to describe the protocols as well as the encoding rules as captured in the code generators. Consequentially, today the development of a data-marshaling layer for a new protocol takes much less time, typically less than a month. The productivity benefits are large, as often these routines are being developed concurrently with the development of the protocol standards. It is common for the length of a field to change or for modifications to be made to the encoding rules. These changes would lead to a total rewrite of the code had it been written by hand, as any bit-level optimization (such as the typical power-of-2 shortcuts) would have been voided by the insertion of additional bits.

PDU marshalling code developed following the described model-driven engineering process has been shipped in a number of Motorola infrastructure network element and subscriber device products, capturing different protocols (such as GSM 4.08, Abis, XBL, TETRA, and 3GPP). Internal tools for the development of PDU marshalling code have been refined to a level of maturity such that generated code consistently is superior to hand-written code in its performance and quality.

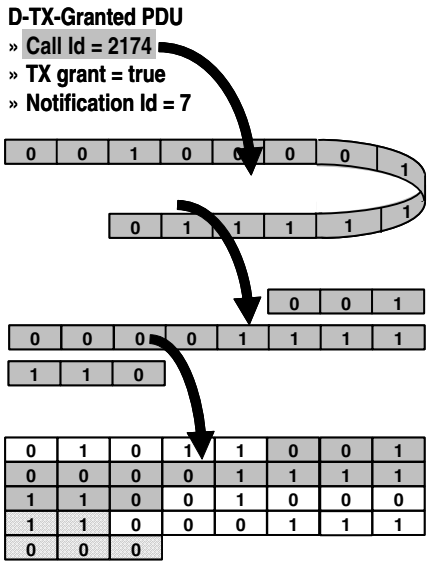

Figure 2. Encoding of Downlink-transmit-granted

3.2. Model-Driven Engineering Rollout

The model-driven engineering vision has been realized in a number of Motorola business units in small steps: In

1989, a design simulation environment for a proprietary design notation was developed and piloted in development projects. When later, in 1991, the first commercial simulation tools became available for a standardized notation (SDL), development teams began migrating to this standard. In 1992, for the first time the complete software for a real-time embedded Motorola product (a pager) was generated from high-level designs, without relying on any handwritten code. It was not until 1994 that the first commercial code generation tools with similar capabilities became available. Subsequently, several Motorola business units adopted design simulation as a new development paradigm. In 1998, the first shipping Motorola products were automatically derived from high-level design specifications (a base station for the TETRA radio communication system and a base site controller for a telecommunications network). The subsequent years saw a steady increase in penetration of model-driven engineering, as legacy products were gradually replaced by newly developed network elements.

This experience also revealed that the standard notations deployed had shortcomings that limited the applicability of code generation. As a response, in 1999, an enhanced version of SDL was adopted supporting language elements required by engineering teams. In 2003, the latest release of UML was adopted, integrating the lessons learned from SDL deployment.

4. Benefits

The benefits afforded by model-driven engineering are productivity and quality improvements. These benefits come from various sources:

- Design models are easier and faster to produce and test
- Labor-intensive and error-prone development tasks are automated
- Design effort is focused on applications, not on platform details
- Reuse of designs and tests between platforms or releases is enabled
- Design models can be verified through simulation and testing
- Design models are more stable, complete, and testable
- Standardized common notations avoid retraining of engineers
- The learning curve for new engineers is shortened

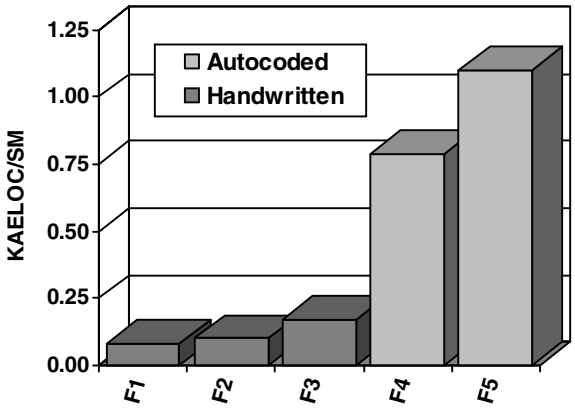

Figure 3. Productivity improvement for network element applications

4.1. Productivity

Pushing much of the development detail into the code generator allows designs to be more abstract, which results in designs that are easier to produce and easier to show correct.

There are fewer inspections required to ensure the quality of the developed code than using conventional development. On average, developers rely on three inspection cycles instead of four cycles when compared to following the conventional process. In addition, inspection rates are higher and have increased from 100 source lines per hour to in between 300 and 1000 source lines per hour. Thus, not only are fewer inspections required, but also the remaining inspections are much more efficient.

Code automation on average results in a five-fold increase in the number of source lines of code produced per staff months over the development life cycle. The effort spent in the design phase increases, but this is more than made up by the dramatic reduction in coding effort.

Code generators have reached a level of maturity that effectively no errors are being introduced into the resultant code. For example, over the last three years, only one defect was detected in our marshalling code generator. Subsequently, no effort has to be expended to correct coding defects.

Figure 3 shows the productivity improvements (as measured in assembly equivalent lines of code produced per staff month) in the development of several features on a base site controller (a core network element). The chart compares the productivity rates achieved using the conventional life cycle with those using model-driven engineering. Figure 4 shows effort reduction (in terms of staff days) during

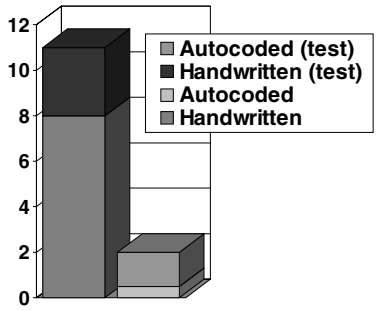

Figure 4. Productivity improvement for data marshaling

the development of a typical protocol data marshaling generator (the lighter shaded areas indicate the testing effort).

Automation of labor-intensive and error-prone development tasks results in additional productivity improvements. We have seen a dramatic reduction in the turn-around time for fixes during process test execution. A process test cycle involves: (i) fixing the defect in the model (rather than the code), (ii) writing a test case, (iii) generating the corrected code, and (iv) automatically executing the complete regression test suite (which in this particular example consisted of over 10,000 tests). As shown in Figure 5, the test cycle across four releases of network features has been reduced from 25-70 days to 24 hours.

Similar results can be observed in box and system tests. In many systems, over 90% of the tests are automated using TTCN scripts, which has led to a 30% reduction in box-test cycle time.

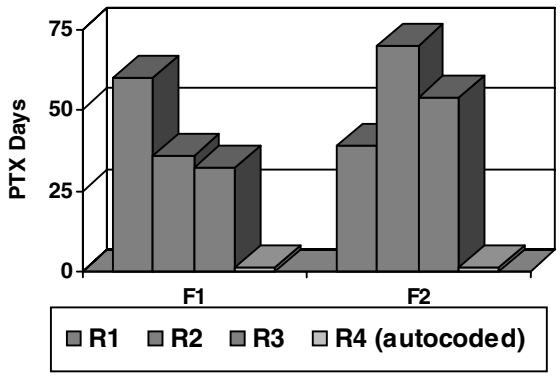

Figure 5. Productivity improvement in process test cycle

4.2. Platform Targeting

Model-driven engineering removes the need for embedding platform characteristics and domain detail into the designs (again impacting productivity and quality). Following the conventional process, much irrelevant information—irrelevant from the point of view of system functionality—had to be kept in the design document to ensure that it be considered during coding. This extraneous information negatively affects productivity and quality.

For example, an important feature of a high-availability middleware layer developed by Motorola is the ability to journal ongoing transactions, allowing a computing node to recover from failure and continue execution at the point where a run-time fault occurs. The code necessary to perform the journaling should not be captured in the design (as it is not part of the functional requirements of the application) but should instead be added by the code generator. Such an approach not only keeps the designs abstract and easier to produce and verify, but also allows experimentation with different levels of journaling granularity to find the right balance between availability and performance.

Avoiding embedding implementation detail into design enables rapid retargeting of applications to different platforms. A given design can be moved from one platform to another without affecting the design. As an example, we moved a base site controller application from a rack of MC6809 cards with distributed memory to a shared-memory computer. All changes required to the software architecture were performed by the code generator. In another example, an application was migrated to a lower-cost platform, where the code generator had to provide SysV message queues transparently to the application and generate code in a manner so as to avoid deadlock problems (on the cheaper system, the same thread could not acquire the same mutex repeatedly). The effort was 10 staff days to capture these differences in the code generator as compared to an estimated 80 days for hand porting the code.

Incorporating platform specifics into the code through a code generator also enables engineers to experiment at a low cost with alternative implementation strategies, software architectures, or hardware choices.

Finally, the separation between platform-specific information and application information supports more efficient team structures. Platform experts capture their knowledge in code generators, and the application development teams can focus exclusively on the design and testing of new applications.

4.3. Quality

The models required as input for code generation are more complete and can be verified through simulation (or

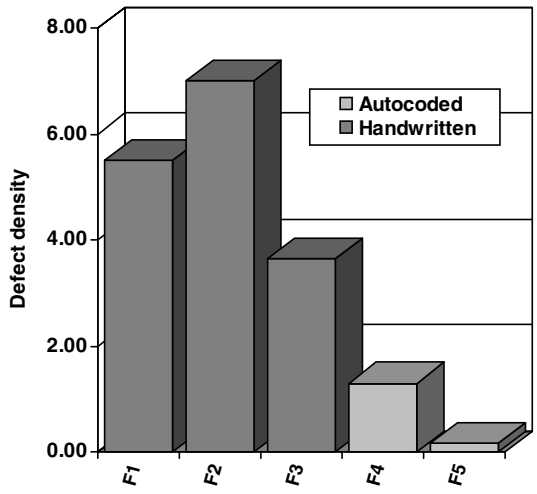

Figure 6. Quality improvement for network element application

other techniques), resulting in significant quality improvements. The quality impact of model-driven engineering is dramatic and almost guaranteed. It is largely a consequence of the increased phase containment effectiveness enabled by semantically precise and operationally interpretable specifications.

Motorola data shows that simulation is about 30% more effective in catching defects than the most rigorous Fagan inspections. This is true for both overall faults and serious faults. Based on this fact alone (assuming that the code generator does not introduce additional defects), we can expect a 3X reduction in defects, which is also borne out by the data: In Figure 6, from the development of features on a network element, we see defect reduction rates well beyond that number. Applications have recently been developed with zero design defects.

Approximately 50% of defects are requirements errors. These are typically the hardest errors to find, and thus also the costliest. Working with models of requirements enables mathematical techniques to be applied to these models, which enables detection of these hard-to-find defects. In the telecommunications domain, many such errors are due to concurrency pathologies, i.e., they result from unforeseen interactions of concurrently executing system components. We have developed techniques based on theorem proving and realized these in tools that detect such situations. Each project that has leveraged these techniques has demonstrated that a substantial number of requirements defects that had previously escaped to later development phases can be discovered early.

Figure 7, which shows percentage of total defects found over time, illustrates that faults are found much earlier following model-driven engineering than with the conventional process. Finding defects sooner is significant, since it is much cheaper to fix defects earlier. Boehm has reported an exponential increase in the cost of fixing defects with the distance between where a defect is sourced and where it is discovered. Finding double the errors in the design phases as shown in Figure 7 translates into large cost savings.

The availability of tools operating on design notations encourages convergence on standard common notations with the associated benefits of sharing expertise and projects between different development teams. The learning curve for new engineers to get familiar with a product has been substantially shortened. When a new engineer studies the application models, domain knowledge is exposed rather than obfuscated in the product code. Consequentially, the time required for a new developer to acquire sufficient domain knowledge to become productive has been shortened by 2-3X.

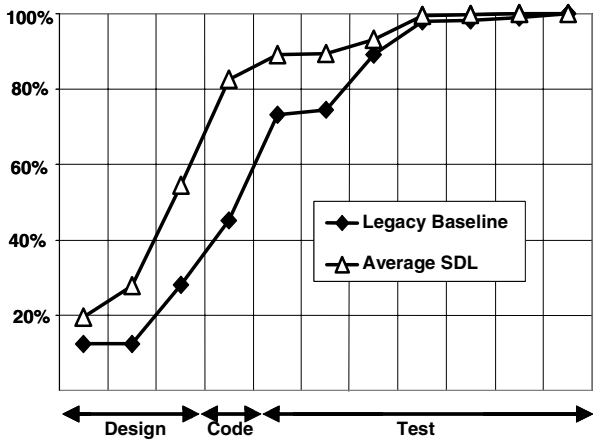

Figure 7. Fault discovery rate across the development life cycle

5. Penetration and Applicability

Deployment of model-driven engineering varies substantially between product teams. Figure 8 shows the penetration across a the components of a release of a telecommunications system, as of 2002. The size of each chart indicates the relative amount of effort that went into producing the corresponding component. The bars, from left to right, indicate the percentage of software that has been developed leveraging model-driven engineering in design, design verification, code generation, and box test, respectively. For ex-

ample, the base site controller development team has used design modeling and code generation for more than 70% of their application code, but has not used simulation to verify the designs. The team developing the surveillance gateway used design modeling, simulation, and code generation for more than 80% of their application code, and the developers were able to drive their box tests from design models as well. On some system components, the code generation rates are still lower due to the large amount of legacy code present.

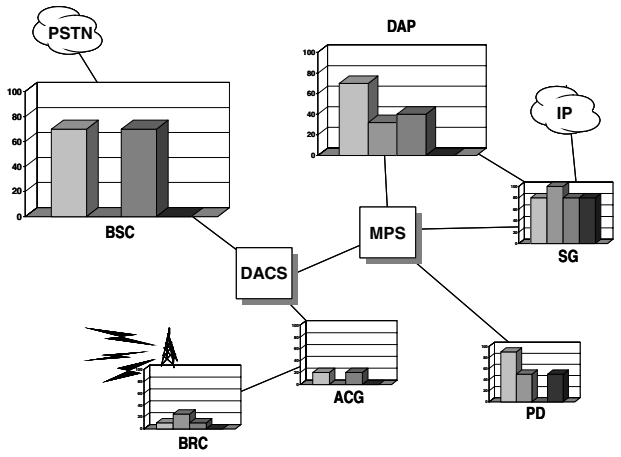

Figure 8. Deployment of model-driven engineering across network element development

We have analyzed the source code for each of the system components of a release of a telecommunications system, categorized the modules of source code by the design elements from which they are typically derived, and grouped the code into four buckets, as shown in Figure 9:

- Code that is specified by state machines or similar mechanisms (such as decision tables or activity graphs)
- Code that is highly algorithmic in nature or expresses data manipulations
- Code that is low level and often not captured in detail in designs
- Code that is described by other means, such as GUI layout or database design tools.

The size of each pie reflects the relative effort that went into developing the particular network element. For example, 56% of the code on the base site controller (BSC) can be characterized by state machines. This software is the main control logic of the application, routing calls between base stations (similar to cells in a mobile telephony system) and the mobile switching system. About 39% of the code is characterized by computation, comprising the packing and unpacking of protocol data units and the evaluation of signal quality to determine whether a call should be handed over between base stations. About 5% concerns the interface to the transcoders and is usually stubbed out in the designs.

The distribution of the four categories varies between network elements. Our experience is that all of the state-machine oriented code can be derived from designs as can most or all of the algorithmic code, depending on the availability of a suitable action language in the design notation or domain-specific notations such as those described earlier for PDU marshaling code. The low-level code is unlikely to be generated automatically. Mileage in the "other" categories varies, but this code comprises a relatively small percentage of the overall application.

Rolling up the data for various telecommunication systems reveals that the potential for model-driven engineering is at least 73%, but may go as high as 96%. The individual percentages are less important than the message: a significant portion of a telecommunication system is amenable to code generation from high-level designs.

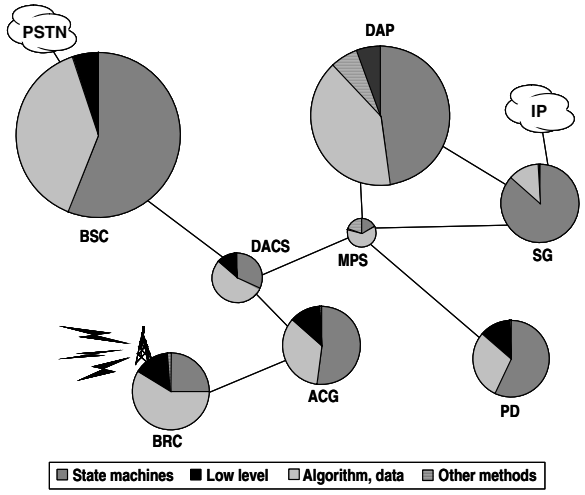

Figure 9. Automation potential across network element development

6. Trustworthy Computing Systems

There are several areas of model-driven engineering that have direct applicability to trustworthy computing systems.

First, coding practices that are deemed insecure or unreliable can be categorically eliminated. That is, coding defects that are prevalent in hand-written code do not exist

in automatically generated code. This means that inadvertent changes to the functionality represented in the models will not be introduced. Early detection of model defects or pathologies is only as useful as the reliability of the downstream steps in the development process, so elimination of a large source of human errors is of immediate benefit.

Second, coding policies related to trustworthiness, reliability and security can be systematically added to the generated code. For example, special error handling for memory-allocation failures can automatically be added to all memory allocation requests, and the prevention of buffer overflow during the decoding of an inbound PDU can be applied to all decoding functions. If a policy changes, that change can be applied to all such situations across all of the generated code without the fear of having missed some cases. These policies also do not lead to the models becoming cluttered. Such crosscutting concerns can be uniformly applied during the transformation of the models into code.

Third, typical problems that lead to trustworthiness concerns can be detected early during model analysis and simulation. This capability is enabled by the precise semantics of the modeling language. Model analysis can detect generally insecure situations such as the presence of race conditions or nondeterministic behavior in the model, assignment of one variable to another when the variables have overlapping but non-equivalent ranges, decision statements where not all of the possible values are covered by branches, etc. It is also much easier to prove properties about a model than it is to prove them about the derived implementation.

Forth, trust issues related to the underlying platform do not have to be captured in the model itself. This separation of concerns has three distinct advantages: (i) the functionality expressed in the model is not tied to a specific platform, allowing ease of moving the application to lesser or more secure platforms, (ii) the models themselves are not complicated by the addition of the trust concerns not related to the application, and (iii) the expertise associated with the application can and should be separate from the expertise related to the trustworthiness of the underlying platform.

Finally, the development effort saved by using model-driven engineering can be applied to the issues that generally are left to the end, if ever (e.g., failure-mode detection and recovery). That is, the productivity improvements realized through the use of model-driven engineering mean that more time can be spent on the "rainy day" scenarios than would usually be available during conventional development. Development organizations typically leave such efforts to whatever time is left after the "sunny day" functionality is accounted for. This often leads to failure modes related to exception cases, which is one of the largest areas vulnerability in software systems.

7. Summary

In this paper, we have summarized the benefits that Motorola has obtained from the deployment of model-driven engineering in the development of trustworthy computing systems, in particular, network elements for telecommunications systems. The benefits afforded by model-driven engineering result in both quality improvements and productivity improvements. In the development of telecommunication systems, a large portion of developed software, in our estimate at least three quarters of the total software, is amenable to leveraging model-driven engineering techniques.