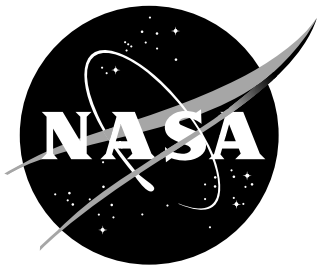
NASA/TM-20200012040



Guidebook on Model-Based Software Engineering and Auto-Generated Code

Katerina Goseva-Popstojanova
West Virginia University, Morgantown, WV 26506, USA

Johann Schumann
SGT, NASA Ames Research Center, Moffett Field, CA 94035, USA

Noble Nkwocha
NASA Independent Verification & Validation Facility, Fairmont, WV 26554, USA

Matt Knudson
NASA Ames Research Center, Moffett Field, CA 94035, USA

June 2018

NASA STI Program . . . in Profile

Since its founding, NASA has been dedicated to the advancement of aeronautics and space science. The NASA scientific and technical information (STI) program plays a key part in helping NASA maintain this important role.

The NASA STI Program operates under the auspices of the Agency Chief Information Officer. It collects, organizes, provides for archiving, and disseminates NASA's STI. The NASA STI Program provides access to the NTRS Registered and its public interface, the NASA Technical Reports Server, thus providing one of the largest collections of aeronautical and space science STI in the world. Results are published in both non-NASA channels and by NASA in the NASA STI Report Series, which includes the following report types:

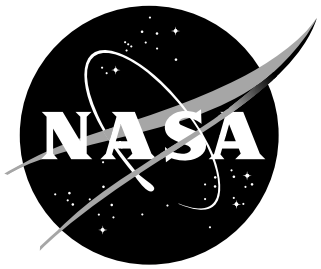
- TECHNICAL PUBLICATION. Reports of completed research or a major significant phase of research that present the results of NASA programs and include extensive data or theoretical analysis. Includes compilations of significant scientific and technical data and information deemed to be of continuing reference value. NASA counterpart of peer-reviewed formal professional papers, but having less stringent limitations on manuscript length and extent of graphic presentations.
- TECHNICAL MEMORANDUM. Scientific and technical findings that are preliminary or of specialized interest, e.g., quick release reports, working papers, and bibliographies that contain minimal annotation. Does not contain extensive analysis.
- CONTRACTOR REPORT. Scientific and technical findings by NASA-sponsored contractors and grantees.
- CONFERENCE PUBLICATION. Collected papers from scientific and technical conferences, symposia, seminars, or other meetings sponsored or co-sponsored by NASA.
- SPECIAL PUBLICATION. Scientific, technical, or historical information from NASA programs, projects, and missions, often concerned with subjects having substantial public interest.
- TECHNICAL TRANSLATION. English-language translations of foreign scientific and technical material pertinent to NASA's mission.

Specialized services also include organizing and publishing research results, distributing specialized research announcements and feeds, providing information desk and personal search support, and enabling data exchange services.

For more information about the NASA STI Program, see the following:

- Access the NASA STI program home page at ***http://www.sti.nasa.gov***
- E-mail your question to help@sti.nasa.gov
- Phone the NASA STI Help Desk at 757-864-9658
- Write to:
  NASA STI Information Desk
  Mail Stop 148
  NASA Langley Research Center
  Hampton, VA 23681–2199

NASA/TM-20200012040



Guidebook on Model-Based Software Engineering and Auto-Generated Code

Katerina Goseva-Popstojanova
West Virginia University, Morgantown, WV 26506, USA

Johann Schumann
SGT, NASA Ames Research Center, Moffett Field, CA 94035, USA

Noble Nkwocha
NASA Independent Verification & Validation Facility, Fairmont, WV 26554, USA

Matt Knudson
NASA Ames Research Center, Moffett Field, CA 94035, USA

National Aeronautics and
Space Administration

Ames Research Center, Moffett Field, CA 94035
West Virginia University, Morgantown, WV 26506
NASA IV&V Facility, Fairmont, WV 26554

June 2018

Acknowledgments

This guidebook is a result of the research work funded by the NASA Software Assurance Research Program (SARP) in FY 2016 and FY 2017.

Executive Summary

Model-based Software Engineering (MBSwE) is a powerful paradigm to develop mission- and safety-critical code for NASA missions. Despite obvious advantages and numerous tools that are to support MBSwE, many intricacies and potential issues can endanger the budget and schedule of the software project and carry the risk of producing code of low quality and reliability. This guidebook is focused on MBSwE and Auto-generated Code (AGC) with a goal to provide a comprehensive overview of the field and popular tools, make the reader aware of important issues, provide practical guidelines for selection of suitable processes and tools, describe the modeling-standards, how to deal with synchronization and maintenance issues, and how to set up a powerful, tool-supported verification and validation process. This guidebook was developed in support of the NASA Software Assurance Research Program (SARP) and draws from a NASA and industry-wide survey on this topic, and case studies based on two NASA missions.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The use of models in Software Engineering has become more and more popular. Supported by a plethora of powerful tools, nowadays complex, and often highly safety-critical software is not being implemented anymore, but automatically generated from models.

Such a drastic alternative in the way of developing software has substantial influence on many issues that are critical for a successful process. Major topics include, among others, verification and validation, synchronization and maintenance of models and generated artifacts, as well as selection of suitable tools.

Within the project funded by the NASA Software Assurance Research Program (SARP), we carried out a large survey [1] on the use of model-based techniques and automatic code generation within NASA missions and in different industry domains, worldwide. The main goals of the survey were to: (1) assess the current state-of-the practice of using MBSwE and AGC, (2) identify and quantify benefits and challenges, and (3) explore the software assurance practice of models and auto-generated code. The survey had a total of 114 respondents who have used MBSwE on their projects. The questionnaire consisted of 46 questions, both multiple choice and free-form. Respondents were not required to answer all survey questions. The mean and median number of questions answered by the 114 respondents were 30 and 39, respectively. To assure that respondents' answers are not biased, the survey was anonymous. Among those that voluntarily provided information about their affiliation, many respondents are affiliated with NASA, including GSFC, MSFC, JPL, Stennis Space Center, and NASA Independent Verification and Validation (IV&V) Facility.

We also conducted empirical case studies based on two NASA missions – one ongoing [2] and another under development [3]. These case studies were focused on exploring the approaches to MBSwE and AGC used by the NASA missions, the SWA practices used by the missions, and empirical analysis of developers' and IV&V bug reports.

The mission's flight software of the first case study [2] is a combination of handwritten code and AGC developed by two different approaches (one based on state chart models created in MagicDraw and the other based on input from specification dictionaries). Approximately 18% of the mission code was auto-generated using these two approaches. 56% of the total source code was newly developed and the rest was either re-engineered or reused. The empirical analysis of the software bugs was based on 380 closed bug reports created by software developers' and 450 closed Technical Issue Memoranda (TIMs) created by IV&V analysts, which hereafter will also be referred to as bug reports.

The software system under study for the second case study is a Flight Software com-

ponent responsible for the Command and Data Handling [3]. The mission used MBSwE approach supported by the Rational Rose development suite. The code was generated by Rational Rose RealTime using the developed UML models together with handwritten code associated with them. According to the developers, these handwritten parts constitute approximately 65% of the software code, that is, 35% was auto-generated. The empirical analysis of software bugs was based on 770 closed bug reports created by software developers' and 178 closed bug reports created by IV&V analysts.

Based upon the results of the survey [1], the two case studies [2], [3], an architecture design [4], as well as a previous survey conducted in 2006 [5], this document aims to provide guidance for the software practitioners, software assurance specialists, and project managers of software-oriented projects who have to decide if to adopt a MBSwE paradigm and to become knowledgeable about background, tools, and practical challenges and benefits of MBSwE and AGC.

Although the observations and recommendations in this guidebook are based upon our survey, two case studies and are drawn from collaboration and discussion with multiple NASA missions, they are definitely not comprehensive and should be considered as guidelines only and not as set in stone.

1.1 Goals and Structure of this guidebook

The main goal of this guidebook is to provide the reader suitable information and recommendations on the topics of MBSwE and AGC. In particular, this guidebook addresses the following questions:

- *What?* What is MBSwE? What is automatic code generation? For which processes and application areas can MBSwE and AGC be used?
- *How?* We focus on popular tools, the purpose of use of MBSwE and AGC, and the use of modeling and coding standards.
- *How to do verification and validation of models and AGC?* We focus on the tools, processes and standards used for Verification and Validation (V&V) of models and auto-generated code.
- *How to synchronize and maintain the models and AGC? How to handle AGC?* The answers to these questions are relevant not only throughout the development process, but also after the deployment.
- *What types of bugs are found in models and auto-generated code? How are the fixes of these bugs distributed?*
- *What are the benefits and challenges of using MBSwE and AGC?* Here we focus on the effect of the MBSwE and AGC on productivity, maintainability, quality, transition from traditional software engineering development to model-based development, and learning efforts.

Last but not least, we synthesized the most important lessons learned based on the research work done by our SARP funded project in a list of recommendations that are expected to be helpful to the software practitioners, software assurance specialists, and project managers of NASA missions that use or plan to use MBSwE and AGC.

The rest of the guidebook is organized as follows. In Chapter 2 we present the background on software development processes, models in SW engineering, and automatic code generation, and list a number of tools that are, according to our survey, popular in this domain. Chapter 3 focuses on the tasks of verification and validation (V&V) in an MBSwE environment with AGC, and presents a model-based V&V architecture that illustrates the specifics when dealing with V&V in MBSwE. We also present a number of popular V&V tools, both for the analysis of models, as well as for the analysis of AGC.

A MBSwE process has to deal with multiple models and software artifacts, some of which are generated automatically. This raises the important question on how to synchronize those artifacts with the models, on how to maintain models and other artifacts, and how to handle automatically generated code. These topics are discussed in Chapter 4.

Chapter 5 is focused on software bugs in MBSwE and presents detailed results based on the analysis of bug reports of two NASA missions.

In Chapter 6, practical considerations about MBSwE and how to set up a MBSwE project are discussed. Chapter 7 presents the benefits and challenges of MBSwE and AGC based upon responses to our survey, as well as upon the quantitative and qualitative findings of our two case studies.

Finally, Chapter 8 summarizes the main topics of the guidebook and provides practical recommendations.

Chapter 2

MBSwE and AGC: Basics and Popular Tools

Model-based Software Engineering (MBSwE) can be considered as a specific way of doing Software Engineering. It affects all phases of the software process, tools, best practices, verification and validation (V&V), as well as software maintenance and reuse. In this Chapter, we focus on specific topics important for the development of models and AGC.

2.1 Software Development Processes

A software (development) process defines when and in which order the main steps of software development are executed. These main steps include all steps of the *lifecycle* of the software: requirements, design, implementation, testing, deployment, and maintenance. For larger projects with numerous developers and safety-critical application areas, the adoption and use of a suitable lifecycle model and software process is extremely important. This is true for traditionally developed software, as well as for model-based development.

There are numerous models for software processes and it is beyond the scope of this guidebook to discuss them individually. Rather, Figure 2.1 gives a graphical overview of several popular lifecycle models.

Our survey [1] revealed that many MBSwE projects (95 respondents) actively use one of the popular lifecycle models. Most popular lifecycle models included Waterfall (34%), Agile (30%), Rapid Application Development (12%), and Spiral [6] (11%). The V-model, which is strongly related to the Waterfall model and arranges the tasks in such a way that verification and validation can be clearly separated, was used by 8% of the respondents. Here, the CENELEC V model and the MIL-STD-498 model were mentioned repeatedly.

Although it seems that most life-cycle models can be used for the traditional development of software as well as for MBSwE software, care must be taken, which lifecycle model and process to choose. This topic is further discussed in Section 6.1.

2.2 Models in Software Engineering

The notion of a "model" in Software Engineering is very broad. Probably coming out of the area of model-based design (MBD), it concerns mathematical and visual models used to address the problem of producing (complex) pieces of software.

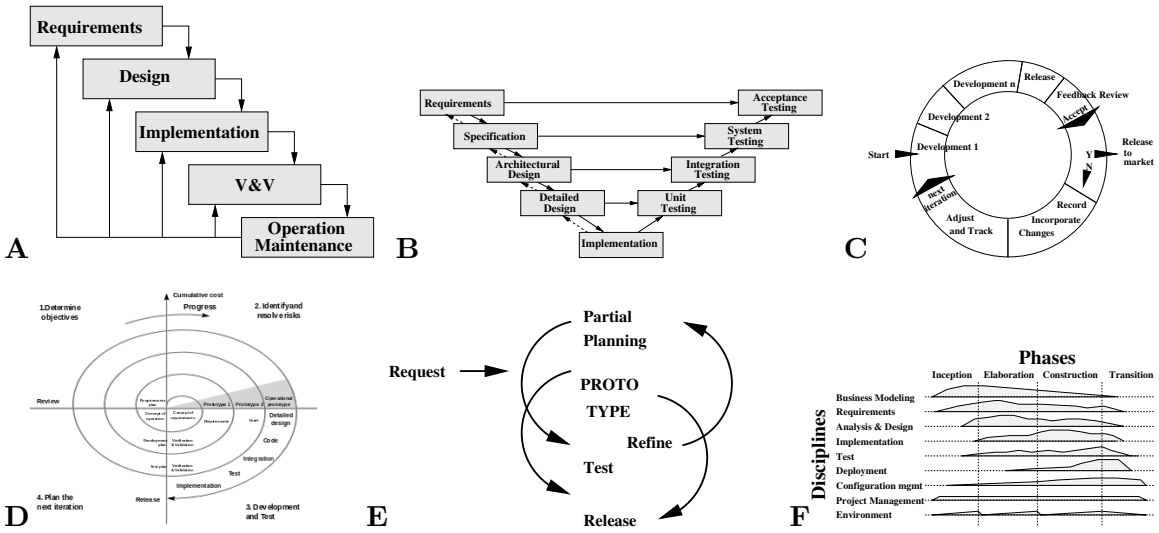


Figure 2.1: Popular life-cycle models: A: waterfall model, B: V-model, C: Agile model, D: spiral model [7], E: Rapid Development application model, and F: Rational Unified model

In general, a "model" can be considered as an artifact that describes some aspect of the software, the plant or its environment, the software process, or its requirements in an abstracted, often graphical manner. A wide spectrum of different modeling paradigms and tools exist. Typical examples include the Unified Modeling Language (UML) [8], which is a graphical representation used to describe the static and dynamic structure of an object-oriented system as well as intended "Use Cases". On a somewhat lower level, Mathworks' Simulink [9] is a graphical environment for modeling of dynamical systems using hierarchical block diagrams.

Figure 2.2 shows a typical "architecture" of a project done with MBSwE. Starting with the requirements, one or more models are designed and constructed. This is usually done using a modeling tool as discussed in section 2.3. Then an automatic code generator (ACG) produces production code from the model(s). This generated code has to be integrated with (manually written) legacy code and must interface with a Middleware (if applicable) and the operating system OS. A traditional compiler/linker tool chain then produces the binary code, which is then executed on the target hardware.

2.3 Popular Tools for MBSwE

In this section, we list a number of popular tools that are used in MBSwE. This list is mainly based upon responses received from participants of our survey [1] and is necessarily incomplete. Typically, tools for MBSwE can be characterized along the following features:

Application Area. Some MBSwE can be used for many different application areas to describe and model systems and their behaviors. Such systems can be physical or chemical systems, mechanical or electrical systems, software systems, or a combination thereof. Other MBSwE tools are tailored toward specific application domains or development paradigms, for example UML for design and modeling of object-oriented software systems.

Model Level. Models can be given in a very high and abstracted way, or they can be

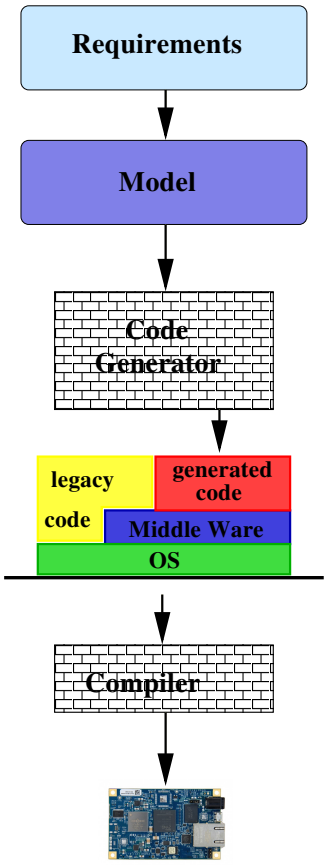


Figure 2.2: Typical architecture for MBSwE

just a graphical representation of code fragments. The semantic distance between the model and the final target artifact often also determines if there is an automatic translation between these representations, for example, the automatic generation of code from a model.

Spectrum. MBSwE spans a wide spectrum ranging from high-level modeling, model analysis, code generation to the generation of test cases and other artifacts (see Figure 2.2).

Whereas some MBSwE tools are just elaborate graphical diagram editors, other tools provide an entire tool chain for modeling, model analysis, model translation, and more. There are even meta-modelling tools, e.g., GME [10] that allow the users to design their own modeling paradigms and tools.

Static vs. Dynamic. Some modeling paradigms are suited to capture the static structure of a system (e.g., a UML class diagram or a component break-down). Other modeling paradigms are designed to capture the dynamic behavior of the system, e.g., a Simulink block-diagram. Often, modeling formalisms contain specific paradigms to handle static and dynamic aspects in a model, as for example, UML class diagrams and UML statecharts.

Model Analysis. In addition to their functionality as (language-directed) graphical editors, many tools come with functionality for the analysis of the model. This can

range from simple syntactical checks to elaborate static model analysis and conformance tools. This functionality is characterized by the fact that a model is actually not executed (or simulated) during the analysis.

Execution/Simulation. Many tools contain functionality to execute the model or to run the model in a simulator mode. For higher efficiency, code might be generated, but usually that code is not the same as the generated target code.

Integration into SW Process. Each use of a software tool must fit into the chosen software process. Whereas some tools are stand-alone tools, other might provide interfaces into software engineering tools. For example, Simulink models can be linked with the DOORS requirements management tool [11].

V&V of modeling tools and Certification. Modeling tools should work as expected. Due to their inherent complexity, tools can have bugs, which can affect the quality of the final target software. Therefore, the quality of the modeling tool is an important decision factor. In highly critical application areas, like Aerospace, standards (e.g., DO-178C [12]) require the certification of the tool to the same level as the software to be developed.

Tool Vendor. There is a wide spectrum of tool vendors and ways of obtaining the tool. COTS tools can be extremely expensive, but there can be Open Source solutions for many tasks in MBSwE. Several commercial tools feature free "community" versions with limited functionality, which often might be sufficient for a given purpose. Often, special purpose model-based tools are developed in-house.

Given this characterization of tools, we now can have a look at popular tools for MBSwE. Below, we list a number of popular MBSwE tools in alphabetical order.

Ameos / OpenAmeos is an UML modeling environment that supports UML 2.0 profiles and MDA based model transformation [13]. It provides MDA-based code generation templates for Java, C++, C, Ada95, and C#. OpenAmeos, which is open-source, is based on the Ameos UML tool from Aonix [14]

AUTOSAR Builder is a modeling and simulation tool for development, simulation, and deployment of embedded systems [15]. It is aimed for the area of automotive systems. The tool suite is based on Eclipse and designed for design and development of embedded systems and software that is AUTOSAR [16] compliant.

Bridgepoint supports model-driven development using executable, translatable UML (xtUML) [17]. This open-source tool provides modeling GUI, a simulated execution environment (Verifier), and model compilers. Commercial versions, like Bridgepoint Pro [18] exist.

DREMS is a software tool for model-based design, implementation, configuration, deployment and management of distributed real-time software [19].

DSpace TargetLink from dSPACE [20] is a tool that generates C code from Simulink/ Stateflow diagrams and is mainly targeted toward the automotive industry.

Eclipse Papyrus UML is a modeling environment for UML based on Eclipse [21]. It also supports SysML 1.1 and 1.4 [22].

Eclipse Xtend is a statically typed programming language that leverages Java's type system and supports lambda expressions, type inference, or operator overloading. It is supported within the Eclipse-IDE [23].

Eclipse Xtext is an Eclipse-based framework for the development of domain-specific languages [24].

Enterprise Architect is a commercial UML design and CASE tool [25].

Generic Modeling Environment (GME) has been developed at ISIS, Vanderbilt University and is a configurable toolkit for domain-specific languages and code generators [10].

iUML is a modeling and simulation tool for the construction and testing of system models using executable UML [26].

MagicDraw is a visual modeling tool for UML, SysML, BPMN and UPDM. It features customizations for domain-specific languages, model decomposition, model transformation, refactoring, and decomposition, as well as template-based document generation [27].

Matlab/Simulink is a model-based design system, which works as a part of Mathworks' Matlab system [9]. Very widely distributed in engineering areas, the hierarchical graphical model features numerous blocks and libraries to design continuous and discrete simulations. Stateflow, an additional package, integrates modeling of hierarchical state machines. Production-level code can be generated using Real-Time Workshop for generic and embedded platform. Mathworks sells additional tools for model analysis ("model analyzer"), testing and testcase generation, as well as static analysis of C/C++ code.

MatrixX is tool chain for control design, consisting of the SystemBuild modeler and AutoCode code generator [28].

Mbeddr is a set of languages and development environment for embedded software engineering [29].

Modelica is an open-source, object-oriented modeling language for the modeling of complex systems [30,31]. Large free libraries facilitate the design of multi-domain models that, e.g., contain mechanical, electrical, hydraulic, or electric subcomponents. Several commercial (e.g., AMESim [32], CATIA [33], or Dymola [34]) and open-source implementations (e.g., JModelica [35] or Openmodelica [36]) exist. Most of these tools provide environments for simulation, optimization, compilation, and model analysis.

merapi Modeling is an eclipse-based UML modeling tool with code generator for C specifically tailored for embedded systems [37].

ObjecTime Developer (OTD) is a software automation tool for generation of production-quality code and is part of Rational [38].

Quantum Leaps – QP Framework is a lightweight software framework for the development of modular real-time systems based upon event-driven actors [39]

Rational Rose is a UML software design tool for visual software modeling and component construction [40].

Rhapsody is a modeling environment based on UML, which is a visual development environment for systems engineers and software developers creating real-time or embedded systems and software [41,42].

ROSLab is a high-level programming language for robotic applications [43].

SCADE is a commercial tool for the model-based design of control software, including software prototyping, verification and validation, and automatic code generation [44]. The code generator has been qualified as development tool for DO-178B/C up to level A and for ASIL level D (ISO 26262).

UPPAAL is an integrated tool environment for modeling, validation and verification of real-time systems modeled as networks of timed automata, extended with data types [45].

Visual Paradigm Eclipse is a commercial UML design tool [46].

The two missions we used as case studies [2], [3] used MagicDraw and Rational Rose, respectively.

2.4 Automatic Code Generation

Automatic Code Generation, in general, concerns the process of generating software artifacts ("code") from other, usually more abstract, higher-level artifacts. This description also includes compilers and program synthesis systems. In a strict sense, a compiler, translating artifacts from one language (e.g., C++) into another one (e.g., binary machine code) can also be considered a code generator. Typically, however, Automatic Code Generators have specific properties:

- Often, the input ("Model") is represented in a graphical way. Typical examples for graphics-based modeling languages are UML or Simulink/Stateflow.

- Automatic code generators usually generate code artifacts in a general programming language (typically C, C++, Java), which then will be used in a regular build process to produce the final executable. Some code generators also generate configuration for hardware realizations of a model, e.g., as a Field Programmable Gate Array (FPGA).

- Automatic code generators can generate numerous artifacts from a single model. For example, in addition to the actual code, configuration files, documentation, or test cases might be produced automatically.

- The semantic gap between input and output representation can be substantially higher than for traditional compilation. Whereas a C compiler, for example, only generates a few machine instructions for each source code line, ACGs often produce large and complex pieces of code for single source constructs. In many cases, entire algorithms (e.g., for table lookup) are automatically generated.

- Automatic code generators can be used specifically for the generation of glue or interface code. Here, a high level model description is used to produce problem-specific code, that, for example, converts data between different representation, or that automatically generates code that implements specific calling conventions of some software module. The code generator avoids tedious and error-prone repetitive code constructs that otherwise need to be entered and edited manually. For example, rpcgen [47] is an early tool that produces code for packing and unpacking of data structures for network communications.

- Automatic code generators can be used specifically for the generation of frameworks and architectures. From a high-level model (often UML), code files are generated that implement a realization of an architecture. These frames then need to be "filled out" with software that is produced manually. For example, a code generator for UML that produces header files and class definitions belongs to this category. Whereas the class and method definitions are generated automatically, the programmer needs to manually implement the "meat" of algorithms and code.

- Code generators can be used specifically for the generation of entire subsystems or software components. These code generators translate a model into a piece of code with a given interface and produce all parts of the software automatically, including declarations, control logic, code for computations, and instantiates entire algorithms. Typical examples include Mathworks Real-time workshop [9] and SCADE [44]. Such tools are often used to generate (large portions) of embedded systems.

- Automatic code generators are, even more than compilers, large and complex pieces of software (perhaps except for the glue-code generators). This means that they usually

  - have numerous options for customization, which makes such tools very flexible, but also introduces potential for making errors,
  - are complex in design (for in-house development), and maintenance,
  - can contain—due to their complexity—substantially more bugs than regular compilers, and
  - can be difficult to set up for a specific project, and can have a steep learning curve.

2.5 Popular Tools for Automatic Code Generation

In this section, we give an incomplete list of automatic code generation tools. Again, most of the system names were provided in the responses of our survey [1].

Ameos / OpenAmeos [13] provides MDA-based code generation templates for Java, C++, C, Ada95, and C# [14].

AUTOSAR Builder is a modeling and simulation tool for development, simulation, and deployment of embedded systems [15]. It is aimed for the area of automotive systems. The tool suite is based on Eclipse and designed for design and development of embedded systems and software that is AUTOSAR [16] compliant.

Bridgepoint supports model-driven development using executable, translatable UML (xtUML) [17] and has model compilers.

DSpace TargetLink from dSPACE [20] is a tool that generates C code from Simulink/Stateflow diagrams and is mainly targeted toward the automotive industry.

Generic Modeling Environment (GME) has been developed at ISIS, Vanderbilt University and is a configurable toolkit for domain-specific languages and code generators [10].

Real Time Workshop (RTW) is the code generator for Mathworks' Simulink and Stateflow [9] model-based design system.

MatrixX AutoCode is a code generator for MatrixX [28].

merapi Modeling is an eclipse-based UML modeling tool with code generator for C specifically tailored for embedded systems [37].

ObjecTime Developer (OTD) is a software automation tool for generation of production-quality code and is part of Rational [38].

SCADE suite is a commercial tool for the model-based design of control software, including software prototyping, verification and validation, and automatic code generation [44]. The code generator has been qualified as development tool for DO-178B/C up to level A and for ASIL level D (ISO 26262).

2.6 Summary

Once we have considered the model-based tools, then our model-based architecture looks as shown in Figure 2.3. Tools for requirements management like DOORS [11] or Polarion [48] often link requirements and model elements. On the various levels of the architecture (*model level*, *code level*, and *hardware level*), popular tools are shown near the appropriate artifacts. Based on our survey [1], the most frequently used MBSwE and AGC tools included:[1] Simulink/ Real-time Workshop (43%), Rational Rose (20%), Rhapsody (20%), MagicDraw (16%), and MatrixX (14%).

Figure 2.3 shows the obvious and prominent purpose of MBSwE– the automatic generation of code. In our survey [1], ACG dominated as a purpose of using MBSwE, with 82%,[2] followed by Design / Prototyping and Simulation / Modeling (each with 74%), testing (45%), and Generating glue or configuration code (27%). The ongoing NASA mission we used as our first case study [2] allowed the developers who used MBSwE to decide the purpose of MBSwE usage. Thus, while some developers used the models only in the design stage, others found it useful to complete the process and produce AGC. The NASA mission under development, which we used as our second case study [3], consistently used the models to auto-generate code.

[1] 92 of 114 respondents answered this question. Some respondents used a combination of two or more tools.

[2] 74 of 114 respondents answered this question. Many respondents listed more than one purpose.

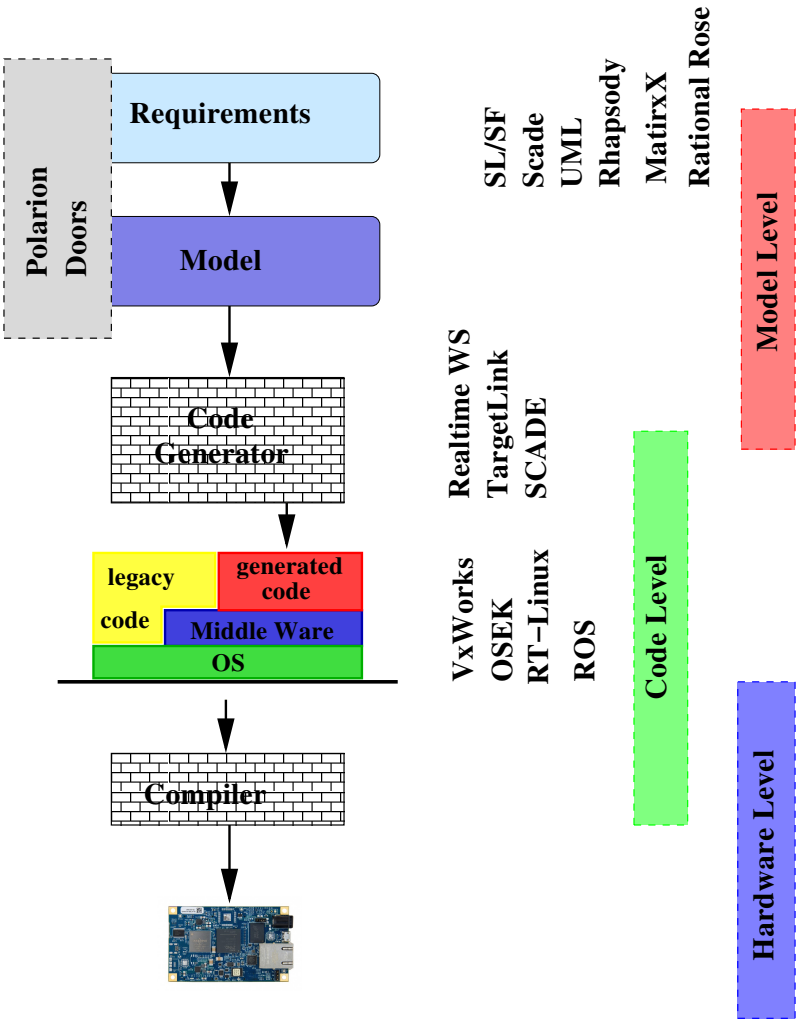


Figure 2.3: Architecture for MBSwE with model-based tools

Chapter 3

Verification and Validation for MBSwE and AGC

3.1 Basics on V&V for MBSwE and AGC

Regardless of the selected software development process, one or more phases exist, when parts (or the whole) of the software must be verified and validated.

Typically, verification consists of answering the question: "Did we implement the thing right?", whereas validation addresses the question of "Did we implement the right thing?". Verification often uses formal analysis and proofs for that purpose, whereas validation is centered around testing.

Obviously, different V&V tasks are typically performed during different phases of the software development cycle. The traditional "V"-shaped diagram (Figure 3.1) shows the typical V&V tasks for classical SW engineering. In Figure 3.1, time proceeds from left to right: starting from the requirements, a specification will be produced, followed by several design phases. After the implementation phase, unit-, integration-, system-, and acceptance testing is carried out. Dashed arrows, going "backwards" indicate Verification tasks. For example, one needs to verify that the specification meets the requirements. Horizontal arrows indicate Validation tasks on the various levels of granularity.

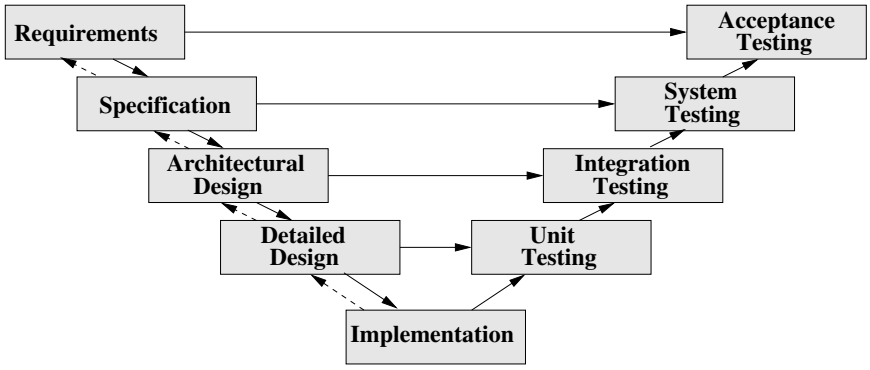


Figure 3.1: The V-model

However, the V-diagram as shown in Figure 3.1 cannot be directly translated into the realm of MBSwE because the newly introduced concept of the "model" and the automated generation of code is not reflected in that figure. Furthermore, not all V&V methods and

tools are applicable for the use in a model-based development environment. For example, a tool for the analysis of buffer-overflow problems is useless if used on a model, which has no notion of an "array" at all.

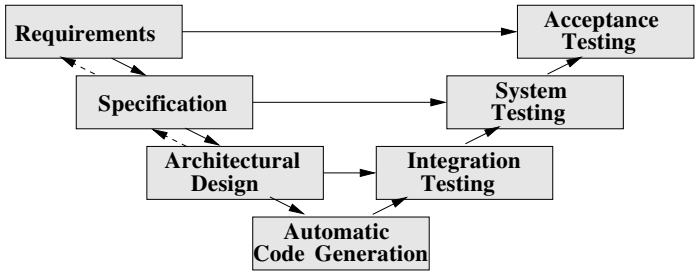


Figure 3.2: Typical V-model when automatic code generation is used

When, in a process with use of AGC, tools are used to generate production code, then the "V" shape has to change, as the "Implementation" phase is missing, as well as parts of the "detailed Design", and "Unit Testing" activities. This results in a shallower "V" shape (Figure 3.2), which also indicates the intended time savings [49]. In addition, the model-based development approach merges most of the boxes on the left-hand side of the V into one large task "Model Development".

If we could assume that the automatic code generator works correctly, then even unit testing would not be necessary. In reality, however, such a strong claim cannot be made but, depending on the code generation tool and process, the amount of unit testing can be substantially reduced. The same may hold for integration and acceptance testing. Of course, V&V on the model level must take place to make sure that the model is correct.

3.2 V&V Architecture for MBSwE and AGC

Due to the fact that model-based tools and automatic code generation tools can reliably perform tasks, which normally would require individual V&V steps (see above), as well as the fact that we now have one additional layer of artifacts, namely "models", we have to design a specific V&V architecture for MBSwE. This is motivated by the fact that within the Safety & Mission Assurance (SMA) sphere of concern, the choice(s) to this question is/are not just fundamental, but crucial. Our architecture separates the V&V concerns into groups dealing with model V&V, and groups about V&V of AGC.

With that architecture in place, it is easier to come up with a configuration of process and tools, which meet the user's goal to efficiently produce highest-quality code. We define our *V&V-architecture* (Figure 3.3) around the MBSwE architecture shown in Figure 2.2. We first discuss the goals of this architecture and then focus on the specific V&V phases, their tools, and interactions.

3.2.1 Goals of the V&V Architecture for MBSwE and AGC

Our V&V architecture needs to address the following questions:

- Does the V&V architecture provide a framework to demonstrate during V&V that "due diligence" has been done and that all necessary (and prescribed) V&V tasks have been performed?

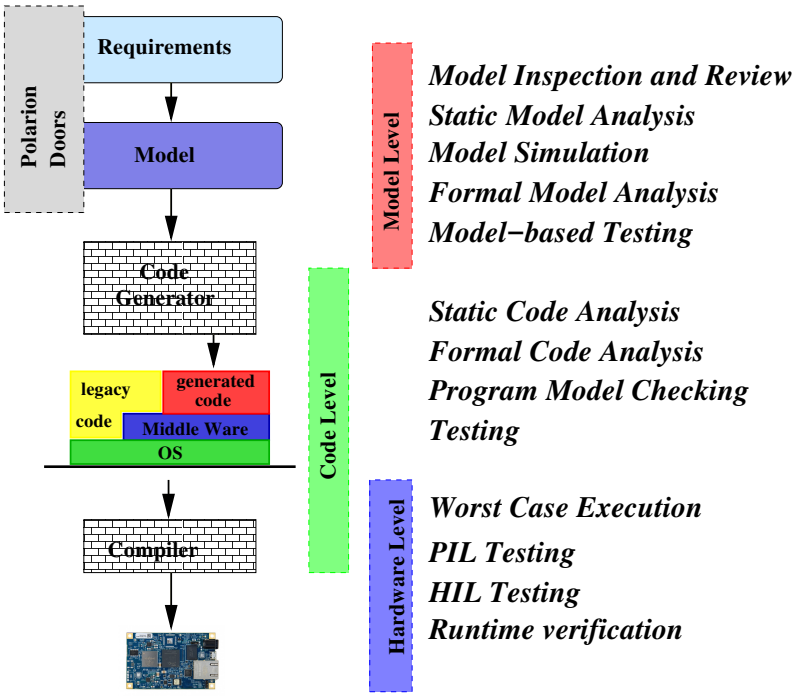


Figure 3.3: V&V-architecture for MBSwE

- Is our V&V architecture effective? This means: do properties that could be shown in the model-level need to be processed again on the code-level (double-work)? and: Are the available V&V tools used as effectively as possible?

- Can the individual results of V&V tasks on model and code level be, at the end, combined into documentation suitable for a certification authority?

Obviously, V&V in an MBSwE environment is extremely important. Our survey revealed that in about 80% of projects, V&V tasks specific to verification of models and tasks specific to AGC were carried out. Besides testing, manual inspection played a dominant role. In the following, we discuss specific approaches for model- and code-level V&V.

3.2.2 Model-Level V&V

V&V tasks on the model-level (red box in Figure 3.3) take the *model* as their basis and perform checks against the requirements and specification. Similar to V&V for traditional software, we can distinguish:

Model Inspection and Review. The model is inspected and reviewed by human experts, usually in a format similar to code review. According to our survey 56% of the respondents perform manual model inspection and review.

Static model analysis. Here, the model is analyzed without actually executing it. For example, tools can analyze the connectivity of sub-components, data types of signals, and many other static properties. During a "model review", details of the model are inspected manually.

Model simulation. If the model is executable (e.g., a Simulink or Modelica model), the model can be executed in simulation on given scenarios. Then, the system outputs are analyzed. Simulation-based analysis is one of the traditional V&V methods on the model level. Typically used in the development of real-time systems, like GN&C systems, numerous properties can be demonstrated on the model level, e.g., response time, robustness, stability.

Formal model analysis. Using formal methods, like theorem proving or model checking, properties are tried to be proven, given the model. Typically, such properties are formalizations of requirements. In case a proof attempt fails, some model analysis tools provide a counter-example that demonstrates what went wrong. Such a feedback can be very helpful for the designers to correct and improve the model.

Model-based testing. The model or parts thereof are exercised (often in simulation), using test stimuli. Test cases can be generated manually (based upon the requirements), but there are also tools that automatically generate sets of test cases for the given model ("model-based test-case generation")

Table 3.1 lists some prominent and well known tools for doing V&V on the model level.

3.2.3 Code-Level V&V

On first sight, code-level V&V of software generated in a MBSwE fashion is not different from code V&V for traditional software – the actual code (written in some programming language) is subjected to analysis, execution, and testing (green box in Figure 3.3). Here again, we can distinguish between typical V&V tasks:

Static code analysis performs analysis of the code without executing it. Several levels of scrutiny exist (e.g., manual code review, check against coding standards, static analysis supported by tools).

Formal code analysis uses logic-based formal methods to prove that certain properties of the code are never violated.

Program Model Checking tries to execute all possible different paths through a program and, for each path checks if a property holds or is violated.

Testing executes the code with a set of test stimuli and compares the outcome of each test run with the desired outcome (oracle).

On the code-level, we typically distinguish between unit-testing and system-testing. Further, test cases can be designed to exercise functional elements of the code, whereas structural test cases try to cause all parts of the code to be exercised. The latter test-regime is important for many software standards (e.g., DO-178B[1]), which require testing to certain levels of code coverage (e.g., MC/DC coverage).

Again, test cases can be produced manually or they can be generated automatically by code-based testcase generator tools.

Tables 3.2 and 3.3 lists some prominent and well known tools for doing V&V on the code level.

[1] `rtca.org`

3.2.4 Hardware-Level V&V

On hardware level (blue box in Figure 3.3), typical V&V tasks include Worst Case Execution Time (WCET) analysis, memory profiling, and testing of low level code. Processor-in-the-loop (PIL) or hardware-in-the-loop (HIL) testing environments are typically used during this stage. Since such V&V activities are far removed from the model-based world, the hardware-level V&V are not discussed in more detail.

3.3 V&V Tools and Practices for MBSwE and AGC

There is a number of tools, both commercial and Open Source that can support V&V of MBSwE and AGC. Table 3.1 lists a number of well-known V&V tools for the model level, and Tables 3.2 and 3.3 for the code level. Most entries of the list originate from respondents of our survey [1]. This list is, by no means, complete and illustrates the large variety of available tools. In these tables, we give the name of the tool and a short description with URL, where applicable. In the Type column, FM indicates a tool that uses a formal methods approach (e.g., model checking, theorem proving, SAT or SMT checking), SA indicates static analysis, and TCG stands for test-case generator. The use of small, in-house tools for doing V&V of AGC seems to be rather widespread, although our survey did not yield any details on their functionality. Typical examples of such mini-tools include generation of test-drivers, testing engines, and report generation. Such tools can be very convenient in practice, because AGC can have a substantially different structure from handwritten code, e.g., a code generator might generate orders of magnitude more files containing code snippets.

Our survey [1] revealed that in the majority of projects, testing was selected method for doing V&V on models (72% of responses) and on code (75%). Figure 3.4 shows the popularity of the different methods. Although the automatically generated code is produced in an entirely different way, it seems standard practice (69%) that the same V&V process is used for manually produced code and AGC. Differences in used V&V methods seem to be attributed to the fact that doing manual code reviews and inspections on AGC is very inefficient, because automatically generated code is essentially unreadable. Both NASA missions used as a case studies in this SARP project (one ongoing and another under development) verified and validated the AGC using the same process as handwritten code [2,3].

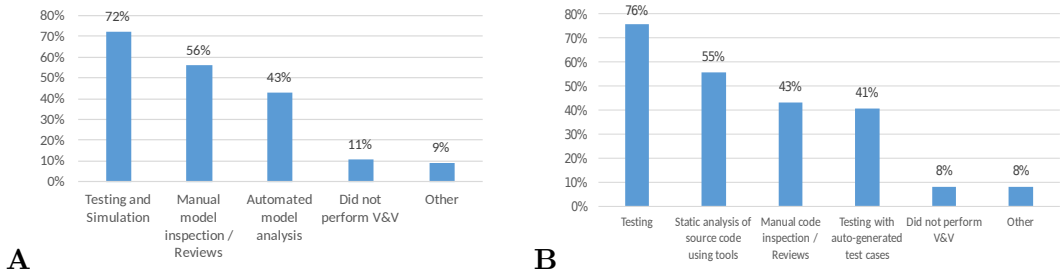


Figure 3.4: V&V methods used for V&V of **A:** models (75 respondents) and **B:** AGC (74 respondents). Multiple selections are possible. From [1].

Name	Type	Description
CoCoSim	FM,TCG	CoCoSim (`https://coco-team.github.io/cocosim`) is a fully automated analysis and compiler for Simulink/Stateflow. It modularly compiles Simulink/Stateflow into Lustre – a synchronous dataflow language. In the backend, it uses different SMT-based infinite state model checkers to validate safety properties. Safety properties are encoded at the Simulink/Stateflow model as synchronous observers. Moreover, CoCoSim allows to generate C or Rust code directly from Simulink/Stateflow using the modular compiler LustreC. `https://github.com/coco-team/lustrec`.
Design Verifier	SA,TCG	Simulink Design Verifier uses formal methods to identify hidden design errors in models without extensive simulation runs. It detects blocks in the model that result in integer overflow, dead logic, array access violations, division by zero, and requirement violations. For each error it produces a simulation test case for debugging. Simulink Design Verifier generates test inputs for model coverage and custom objectives. `https://www.mathworks.com/products/sldesignverifier.html`
Model Advisor	SA	The Model Advisor is a tool in core Simulink that automatically checks your model for some common mistakes. The Model Advisor can analyze the entire model, or a subset of the model. `http://mathworks.com`
Reactis	FM	Reactis plays a number of important roles in a model-based design process. It gives you a set of tools to test, verify, and truly understand your model's behavior and can help you check whether the source code that will ultimately be deployed conforms to the behavior of your model. `http://www.reactive-systems.com`
T-VEC	FM,TCG	The T-VEC Test Vector Generation System is a tool suite for model-based functional test generation. It operates on rigorous system or software models to verify their integrity and automatically generate test cases useful in verifying an implementation of the models. These models are typically developed in modeling tools available from T-VEC or third-party vendors. The tool suite includes the Test Driver Generator for transforming the generic test vectors into test drivers or test scripts for test execution. `https://www.t-vec.com/solutions/tvec.php`

Table 3.1: V&V tools for model level

Name	Type	Description
AbsInt	SA	aiT automatically computes tight upper bounds for the worst-case execution time of tasks in real-time systems. StackAnalyzer automatically determines the worst-case stack usage of the tasks in embedded applications. It directly analyzes binary executables and takes cache and pipeline behavior into account. Astre automatically proves the absence of runtime errors and invalid concurrent behavior in C applications. It is sound for floating-point computations, very fast, and exceptionally precise. The analyzer also checks for MISRA coding rules and supports qualification according to various standards. `https://www.absint.com`
Code Inspector	SA	automatically compares generated code with its source model. The code inspector systematically examines blocks, state diagrams, parameters, and settings in a model to determine whether they are structurally equivalent to operations, operators, and data in the generated code. `https://www.mathworks.com/products/simulink-code-inspector.html`
Code Sonar	SA	CodeSonar, identifies bugs that can result in system crashes, unexpected behavior, and security breaches. It can analyze source and binary code. `https://www.grammatech.com/products/codesonar`
Coverity	SA	develops platform solutions for compile-time source code analysis in C, C++, and Java featuring defect detection analyses. `http://www.coverity.com`
cppcheck	SA	is a static code analysis tool for the C and C++ programming languages. It is a versatile tool that can check non-standard code. The goal is to detect only real errors in the code (i.e. have zero false positives). Open-source. `http://cppcheck.sourceforge.net`
IKOS	SA	IKOS, developed at NASA ARC is a C++ library designed to facilitate the development of sound static analyzers based on Abstract Interpretation. Specialization of a static analyzer for an application or family of applications is critical for achieving both precision and scalability. Developing such an analyzer is arduous and requires significant expertise in Abstract Interpretation.

Table 3.2: V&V tools for code level

Name	Type	Description
Klocwork	SA	delivers source code analysis solutions using static analysis and complete codebase inspection for C++,C, C# and Java. http://www.klocwork.com
LDRA Tool Suite	FM,TCG	provides traceability throughout the SW lifecycle, compliance with coding standards, automatic unit and system-level test, report generation and automation of generation of SW certification evidence http://www.ldra.com
Polyspace Bug Finder	SA	identifies tun-time errors, concurrency issues, security vulnerabilities and other defects in C and C++ embedded software. Can check compliance with coding standards and can trace back results to Simulink models https://www.mathworks.com/products/polyspace-bug-finder.html
Polyspace Code Prover	SA	is a sound static analysis tool that proves the absence of overflow, divide-by-zero, out-of-bounds array access, and certain other run-time errors in C and C++ source code. https://www.mathworks.com/products/polyspace-code-prover.html
SeaHorn	SA	SeaHorn is a software verification framework. The key distinguishing feature of SeaHorn is its modular design that separates the concerns of the syntax of the programming language, its operational semantics, and the verification semantics. SeaHorn encompasses several novelties: it (a) encodes verification conditions using an efficient yet precise inter-procedural technique, (b) provides flexibility in the verification semantics to allow different levels of precision, (c) leverages the state-of-the-art in software model checking and abstract interpretation for verification, and (d) uses Horn-clauses as an intermediate language to represent verification conditions which simplifies interfacing with multiple verification tools based on Horn-clauses. SeaHorn is able to check user-supplied assertions, buffer (under)-overflow, code inconsistency and termination checks. http://seahorn.github.io/

Table 3.3: V&V tools for code level (continued)

Chapter 4

Synchronization, Maintenance, and Handling of Models and AGC

In MBSwE, multiple artifacts can be generated by automated tools. Such artifacts can include generated (production) code, documentation, test-cases, and many more. In this chapter we focus on how the models can be synchronized with those artifacts, how all artifacts in MBSwE can be maintained, and how automatically generated code should be handled.

4.1 Synchronization and Maintenance of models and AGC

As the software life-cycle progresses, it is important to adapt practices related to the synchronization and maintenance of the models and AGC. Here, we briefly discuss the main findings based on our survey [1] and the exploration of two NASA missions [2], [3] used as case studies.

Two thirds of the respondents (i.e., 66%) of our survey [1] stated that the models and the auto-generated code were "always" synchronized throughout the project. Of the remaining respondents 26% synchronized the models and auto-generated code "occasionally" and 8% "never" synchronized the models and auto-generated code. Lack of synchronization between models and AGC led to significant (5%), moderate (13%), and minor (17%) problems [1]. (For 54% of survey respondents this question was not applicable.)

It is important to emphasize that both missions we used as case studies during the course of this SARP project [2], [3] kept the models and AGC always in-sync.

In most cases, both the models and auto-generated code are maintained after the development phase of the project. Based on the results of our survey [1], 84% of the respondents maintained the models and 71% maintained the auto-generated code after the development phase.

One of the challenges of projects that use MBSwE is the fact that the software product is never completely auto-generated; rather it is some combination of AGC and handwritten code. Based on our survey [1], the portion of AGC was almost evenly distributed among the four quartiles. Thus, out of 93 respondents who answered this question, 14%, 20%, 26%, and 24% claimed that 0% to 25%, 26% to 50%, 51% to 75%, and 76% to 100% of the source code was auto-generated, respectively. (16% of respondents did not know how much of the source code was auto-generated.)

The ongoing NASA mission used as our first case study has 18% of the mission code

auto-generated using two different approaches (one based on state chart models AGC-M and another based on specification dictionaries AGC-D) [2]. The fact that the flight software of this NASA mission is a hybrid system (i.e., combination of handwritten and auto-generated code) [2], imposed some maintenance challenges as it required two architectures to be maintained. In addition, developers believed that AGC is likely to be hard to maintain onboard because the whole modeling suite, and thus the models, are not available on the onboard console.

The second case study based on a NASA mission under development [3] used the Rational Rose RealTime to generate the code from the UML models together with the handwritten code embedded in those models. Based on an estimate provided by the mission developers, approximately 35% of code was auto-generated (that is, 65% was handwritten code embedded in UML models).

4.2 How to Handle Auto-generated Code

Always handle Auto-generated Code like a chest of Antique Chinese porcelain: "Do not open, do not touch, handle with care".

Unless the model-based tool generates the entire target software system, the AGC is one or more pieces of code that have to interface with handwritten (newly developed or legacy) code, libraries, middleware and the OS (see Figure 2.2). The interface definition can be part of the model; in many cases, the tool prescribes how the interface looks like. The following are the specific characteristics of AGC, which need to be kept in mind in a MBSwE environment:

- The structure of the ACG can be very compact and unreadable. As the code generation usually aims at high performance, numerous optimizations can yield "ugly" code. Furthermore, variable names in the AGC can be unintuitive, unreadable, or even misleading. Depending on the code generator, variable names can be reused in different contexts and even may vary between different runs of the tool. So, a variable `altitude_0127` might have nothing to do with the altitude in the model.

  Thus, automatically generated code cannot (and should not) be inspected and reviewed manually.

- Because of the complexity of code generation, the synchronization between model and code is an important topic that was discussed in Section 4.1.

- AGC should *never* be modified. A re-generation of code (e.g., triggered by a project build) would overwrite such changes and result in wrong or inconsistent code.

- Interestingly, one third (33%) of respondents of our survey [1] answered that the auto-generated code was modified manually after the development phase. (57% did not modify the auto-generated code manually and for 10% of the respondents the question was not applicable.)

- None of the two NASA missions we used as case studies modified the AGC manually [2], [3]. For the first case study [2], based on the experience with previous missions, both models and AGC were committed in the version control system; once committed, AGC was not changed manually. For the second case study [3], on occasions the generated code may have been modified manually, but these patches were only

temporary workarounds. The change was always propagated to the model elements and synchronized with the generated code in the next build.

- Automatic patches of generated code (e.g., using some scripting editors like sed, awk, perl, or patch) need to be avoided, because internals of the generated code might change without warning. With different tool versions, details of the code generation also usually change.
- The interface to AGC, which is usually prescribed by the tool can require a customized glue code. It is essential that the glue code fully adheres to the documented API of the generated code and never ever should try to exploit undocumented code features.
- Software produced using AGC can contain bugs. Since this issue has widespread implications, we discuss it in details in the next chapter.

Chapter 5

Analysis of software bugs

MBSwE and automatic code generation are used to produce complex, safety critical software, which means that numerous bugs may be found at different lifecycle phases, associated with different software artifacts. Those bugs may be due to requirements (e.g., incorrect, ambiguous, or missing requirements), introduced by the modelers, caused by some misunderstanding about model semantics, or caused by bugs in the actual MBSwE tools and automatic code generation tools. In particular, automatic code generation tools are extremely large and complex pieces of software that, despite all promises by the tool vendors, can generate buggy code.

So, it is not surprising that the vast majority of the respondents (i.e., 83%) of our survey found bugs in models, while fewer, but still substantial number of respondents (i.e., 60%) found bugs in the auto-generated code [1]. This percentages indicate that MBSwE leads to early detection of at least some bugs, but there are bugs still remaining in the auto-generated code.

With respect to the types of bugs found in the models and AGC, as shown in Figure 5.1, the survey respondents indicated that significantly more requirements bugs, design bugs, structural logic bugs, syntax bugs, misleading/wrong documentation bugs, interface and integration bugs, and architecture bugs were detected in models than in the AGC [1]. This observation seems to indicate that

- Based on survey respondents opinions, the MBSwE is serving its purpose of early detection of some type of bugs.
- The bugs found on model level and on code level often belong to different categories. Therefore, the use of V&V tools specifically tailored towards the model level and the code level is justified.

Note, however, that the numbers of responses to the question related to the type of bugs was rather low (i.e., from 11 to 35 respondents for different types of bugs, out of 114 respondents).

In addition to the survey, we conducted empirical studies based on two NASA missions, one ongoing [2] and another under development [3].

For the first case study, based on ongoing NASA mission, the flight software is a combination of handwritten code and AGC developed by two different approaches. The first MBSwE approach used the statechart models created in MagicDraw as input to an in-house developed auto-coder running in the Quantum Framework to generate C code. We refer to this code as Auto-Generated Code - MagicDraw (AGC-M). Note that the parts that

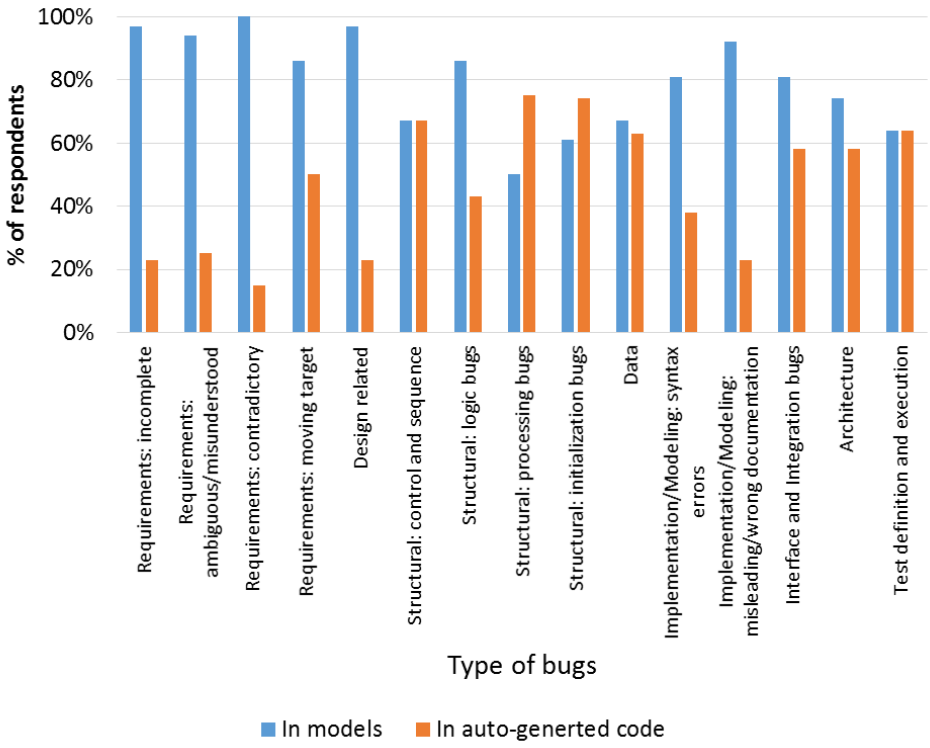


Figure 5.1: Types of bugs observed in the models and auto-generated code [1]. Responses to each part of question ranged from 11 to 35.

were auto-generated based on the statechart models were typically the logic (i.e., engine) portions of that module. The second MBSwE approach led to producing auto-generated code based on input from specification dictionaries. These dictionaries contain command and telemetry data which were exported in an XML based format and then passed to two in-house developed autocoders that produced C source code. These type of AGC is annotated as Auto-Generated Code - Dictionary (AGC-D). Note that the code developed using the second approach was structural only, with no behavioral details. Approximately 18% of the mission code was auto-generated using these two approaches. 56% of the total source code was newly developed (New) and the rest was either re-engineered (ReE) or reused (ReU). The empirical analysis of software bugs and fixes made to fix these bugs was based on 380 closed bug reports created by software developers [2].

We found that

- Most bug reports led to changes in more than one type of file.
- Most fixes required changes to handwritten source files. This was expected as handwritten code made up the largest portion of the code base.
- Less bug reports led to changes in auto-generated source code files. The majority of changes to auto-generated source code files were made in conjunction to changes in either MagicDraw models (.mdxml) or XML files derived from dictionaries (.xml), which confirms that auto-generated code and models were kept in-sync.

The analysis of software bugs at file level appeared to be more relevant than at module level because none of the modules was fully auto-generated. In addition, file sizes were more

Table 5.1: Basic statistics of the number of fixes at file level, by development approach and heritage, split by .c and .h files

Dev approach	Heritage	# of files		Total LLOC		Mean LLOC per file		Total fixes		Median fixes per file		Mean fixes per file	
		(.c)	(.h)	(.c)	(.h)	(.c)	(.h)	(.c)	(.h)	(.c)	(.h)	(.c)	(.h)
AGC-M	New	15	23	5,135	720	342	31	56	66	4.00	3.00	3.73	2.87
	ReE	2	4	231	66	116	16	0	0	0.00	0.00	0.00	0.00
	ReU	0	0	0	0	0	0	0	0	0.00	0.00	0.00	0.00
AGC-D	New	44	44	6,551	3,315	149	75	11	10	0.00	0.00	0.25	0.23
	ReE	34	34	3,286	1,769	97	52	5	3	0.00	0.00	0.15	0.09
	ReU	7	7	285	174	41	25	3	2	0.00	0.00	0.43	0.29
Handwritten	New	128	79	45,317	6,868	354	87	455	216	2.00	2.00	3.55	2.73
	ReE	87	45	26,573	4,170	305	93	265	135	2.00	2.00	3.05	3.00
	ReU	48	26	14,320	1,617	298	62	94	42	1.00	1.00	1.96	1.62

Table 5.2: Basic statistics of the number of fixes per KLLOC at file level, by development approach and heritage, split by .c and .h files

Dev approach	Heritage	Median fixes / KLLOC		Mean fixes / KLLOC	
		(.c)	(.h)	(.c)	(.h)
AGC-M	New	11.83	100.00	14.79	221.19
	ReE	0.00	0.00	0.00	0.00
	ReU	0.00	0.00	0.00	0.00
AGC-D	New	0.00	0.00	2.97	4.97
	ReE	0.00	0.00	1.98	2.17
	ReU	0.00	0.00	10.21	13.79
Handwritten	New	14.87	52.63	22.44	154.93
	ReE	12.39	37.66	28.63	164.28
	ReU	8.89	38.69	20.68	88.91

balanced than module sizes. We found that among newly developed files, AGC-M files had slightly higher median number of fixes per file than handwritten files; AGC-D files were the least fault prone.

At a closer look, we found that header files (.h files) had rather high mean fixes per kilo lines of logical code (KLLOC), which was due to their small sizes. Therefore, we conducted the analysis separately for .c and .h files. The results are shown in Tables 5.1 and 5.2.

As can be seen from Table 5.1, for newly developed .c files, the median number of fixes per file of AGC-M was higher than for handwritten code (i.e., 4 vs. 2 fixes per file). The mean number of fixes per file of AGC-M and handwritten code have comparable values (i.e., 3.73 and 3.55, respectively). Similar observations were made for the newly developed .h files. The AGC-D files (both .c and .h) were the least fault prone, with zero median fixes per file and an order of magnitude smaller mean number of fixes per file for new files. Interestingly, reused AGC-D files had higher mean number of fixes per file than new and re-engineered AGC-D files.

When it comes to the fix density (i.e., the number of fixes per KLLOC), as can be seen from Table 5.2, .h files have significantly higher fix density than .c files, for all statistics. This is mostly due to the fact that .h files have an order of magnitude smaller file sizes that .c files (see Table 5.1). For newly developed files, AGC-M .c files have slightly lower median fix density than newly developed handwritten files, while AGC-M .h files have significantly higher median fix density than handwritten files. The same is true for the mean fix density. The mean values, however, are higher than the median values as they are more sensitive to outliers.

Unfortunately, information on root causes was not available in the Developers' bug

reports.

We also had access to the IV&V issue tracking system and analyzed 450 closed bug reports [2]. However, since the IV&V effort was discontinued due to lack of resources, these bug reports were mostly related to the early life cycle activities, before the AGC was created. Since the analysis of these bug reports is inevitably incomplete, it is not presented here.

Overall, based on the first case study [2] the main findings related to software bugs are as follows:

- The fault proneness of AGC-M files is comparable to the fault proneness of the handwritten code. Specifically, newly developed .c AGC-M files had higher median number of fixes per file and somewhat lower median fixes per KLLOC than the handwritten .c file.

- Among new files, AGC-D files (both .c and .h) were the least fault prone, regardless of the metric used (i.e., median and mean fixes per file, as well as the median and mean fixes per KLLOC).

- Development team found the MBSwE approach based on dictionaries to be much more useful that the approach based on state charts. This qualitative observation based on developers' opinions is consistent with the quantitative observations about fault proneness given above.

Our second empirical study was based on a NASA mission under development [3]. The mission used MBSwE approach supported by the Rational Rose development suite. Handwritten code was associated with the transitions from one state to another in the UML State Transition Diagrams (STDs). Handwritten code was also added via other UML model elements, such as Class diagrams and Capsule diagrams. The code was generated using Rational Rose RealTime. Accordingly to the developers, the handwritten parts constitute approximately 65% of the source code, that is, 32% of the code was auto-generated.

The empirical analysis of software bugs was based on 770 closed bug reports created by software developers and 178 closed bug reports created by IV&V analysts.

Table 5.3 lists the combination of artifacts that the Developers' bug reports were related to. As can be seen, "Implementation Model" was the most common artifact type that bug reports were related to. (Note that here "Implementation Model" refers not only to the corresponding UML model, but also includes the handwritten code associated with it.) Specifically, 72% of Developers' bug reports that had the "Artifacts" field populated were tied only to "Implementation Model", and additional 16% were related to the "Implementation Model" in combination with one or more additional artifacts. (Note however, the value of the "Artifacts" field was missing in 62% of the bug reports.)

The IV&V bug reports explicitly identified the bug (i.e., fault) categories. Their distribution is shown in Table 5.4. As can be seen, the Requirements faults, Implementation faults (i.e., faults found in the code), and Test faults dominated, with 26%, 43%, and 24% of bug reports that had the fault category identified, respectively. Note that for the fault category Implementation (i.e., faults found in the code) the actual root cause may have been associated with the UML model or the associated handwritten code. However, because of the MBSwE development approach used by the mission (i.e., the fact that the AGC and handwritten code are intertwined), further analysis and attribution of the faults to AGC or handwritten code were unfeasible. Therefore, unlike for the first case study [2], in the case

Table 5.3: Combination of artifacts related to Developers' bug reports

Artifacts	# of bug reports
Baseline System	1
Requirements	4
Requirements, Users Guide	5
Requirements, Interface Control Document (ICD)	1
Implementation Model	209
Implementation Model, Requirements	3
Implementation Model, Requirements, Test Model, Test Model Results, Users Guide	2
Implementation Model, Requirements, Users Guide	9
Implementation Model, Design Guidelines	1
Implementation Model, SW Architecture	1
Implementation Model, Test Model	11
Implementation Model, Test Model, Test Model Results	2
Implementation Model, Test Model, Users Guide	3
Implementation Model, Test Model, Version Description Document (VDD)	1
Implementation Model, Tools Requirements	1
Implementation Model, Users Guide	13
Users Guide	20
Version Description Document (VDD)	2
IRD	1
Missing value	480

Table 5.4: Distribution of fault categories of IV&V bug reports

Fault category	# of bug reports
Requirements	47
Design	12
Implementation (found in code)	77
Test	42
Total	178

of the second case study [3] it was not possible to compare the fault proneness of the AGC and handwritten code.

The main findings based on the second case study [3] include:

- The significant number of root causes related to requirements are likely to affect the software regardless of the development approach being used (e.g., MBSwE vs. traditional (handwritten) software development).
- The numbers of bug reports associated with Design reviews in case of Developers' bug reports and found in the Design phase in case of IV&V bug reports were rather small.
- On the other side, the numbers of bug reports attributed to the Implementation model were significant both in the case of Developers' and IV&V bug reports.

It is interesting to note that for both case studies, we found that the distribution of number of fixes follows the Pareto principle, with relatively few software units (e.g., modules

or subsystems) containing the majority of fixes. Specifically, for the first case study 55% of all fixes were associated with only 20% of modules.(The total number of modules is 70.) For the second case study the six most fault prone subsystems out of 28 subsystems (i.e., 21%) were responsible for 65% of all bug reports that had the subsystem field populated. Both these results were based on the analysis of Developers' bug reports and are consistent with our earlier results for another NASA missions [50].

Chapter 6

Setting up Shop

In this Chapter we focus on how to set up a tool supported MBSwE development by addressing the selection of the lifecycle model, processes, and tools, and discuss the modeling and coding standards, and certification of MBSwE and AGC.

6.1 Selection of Lifecycle and Processes

For each project, the selection of lifecycle and process is crucial. This decision depends on many factors: project/program directions, experience in the team, available tools, and application areas. It is also a long-term commitment since a software lifecycle contains post-deployment elements. We therefore do not discuss this selection in general but rather convey relevant information from our survey on how project teams have handled this issue, and if and how the model-based paradigm influences the selection of lifecycle and process.

As discussed in Section 2.1 MBSwE seems to work well together with many well-known software lifecycle models, like Waterfall, Agile, Rapid Application Development, or Spiral. However, there are a few observations with respect to using MBSwE. In contrast to a traditional software project, model-based approaches shift the focus and effort of many tasks during the lifecycle. Typically, the modeling phase is more extensive in a model-based approach, because the model to be developed usually has to contain more details than a typical high-level design. On the other hand, the implementation phase might be much shorter if large portions of the code are auto-generated. In general, MBSwE with AGC tend to behave favourably in iterative processes.

In a model-based process, the development of requirements and the model is often more interwoven. So, initial architectures (which are part of the model or can be derived from it) can be made visible earlier in the process than in a traditional process, where the architectural design often comes just prior to implementation.

The generation of a model takes its time. In particular, if the model has to span large portions of the software system, substantial effort has to be spent, before the model is "ready" to produce some (even initial) code. Therefore, a first "demonstratable" code might be available only later in the process. On the other hand, many MBSwE tools have excellent model simulation capabilities, so initial demonstrations of functionality are easy to set up.

6.2 Selection of Tools

The *early* selection of MBSwE and AGC tools is extremely important. It is much more urgent than tool selection for traditional, manual code development, because early artifacts (requirements, design models, etc.) are already being processed by tools. There is a number of issues in MBSwE that need to be decided early in the process. Most importantly,

- **MBSwE and AGC usually set up an entire tool *chain*.** This means that all individual tools must be compatible with each other.

- **Determine if the MBSwE tool(s) are suitable for the selected SW process and requirements for certification.** For example, if a system has to be developed according to DO-178C, Level A, this means that not only the code (manually developed and auto-generated) has to certified according to that level, but also the tool chain, specifically for this project. In order to avoid pitfalls later, the certifiability of the selected tools and/or the V&V process must be considered carefully. For more details see Section 6.4.

- **Particular care must be taken in determining and deciding languages and dialects.** Typically, AGC tools generate code for a specific version of the target language. That version must be compatible with the used compiler and the programming language and style used for the rest of the project.

- **MBSwE tools and AGC tools are highly complex systems.** Their successful use does not only require careful setup and IT services, but also training. Only if enough training is provided for all involved personnel and if enough experience from prior projects is available, successful model generation and code generation can be accomplished. Otherwise, "modeling styles" and modes of tool use develop over time during the active project, leading to highly inconsistent and hard to understand models.

- **Tool versions and maintenance.** In contrast to standard programming languages and their compilers, which remain stable and constant for a long time (several years), MBSwE and AGC tools are usually actively developed. This means (a) that internal representations and file formats can change, rendering older versions of models unreadable, and (b) new "cool" features are added to the tools.

  In particular for space applications, where the software system has to be maintained and kept operational for many years, a concise planning and fall-back positions are absolutely mandatory. Otherwise, situations can arise that a current version of a tool cannot read in the original models. The old tool, however, requires a version of the operating system that has been considered unsafe and obsolete. Tools, that only run under Windows XP, are typical examples.

  For many projects, the use of Virtual machines can mitigate such problems in a safe manner, and therefore should be considered.

  If new features of a MBSwE tool or AGC tool are "detected" and used by developers, this can lead to similarly severe problems – new versions of the tools might not be compatible with the rest of the tool-chain anymore, and the original tool version is not capable of processing the newer model versions.

In such cases, the adoption and enforcement of a strict *modeling standard* and an early freeze of the tool chain can help avoid such tedious problems.

- **Licensing.** Most MBSwE tools and AGC tools are highly complex commercial products, which require the project to obtain licenses. When planning to use one or more commercial tools, the following issues should be considered:
  - License duration and costs for upgrades: as mentioned earlier, the tool chain for a project must be kept alive for maintenance purposes. Therefore, working tools are necessary. Companies might charge considerable amounts of money for continuous use of their products and regular updates and bug fixes.
  - Licenses for all tools of the tool chain must be kept in sync. A "sorry, license expired" produced by the GUI component after a week-long run of the analysis tool (as happened to one of the authors) can waste a lot of precious time.
  - The number of seats for each tool needs to be considered carefully in order to (a) avoid unnecessary delays in builds or analysis and (b) unnecessary high costs.
  - License costs for additional tasks. For example, if a modeling tool can only *display* the model if there exists a valid license, than *each* external reviewer (if a model-review is requested) must have access to a full license of that tool. That can cause substantial additional costs. The availability of free "readers" can be very helpful. For several major aerospace projects, this had been a major roadblock in using MBSwE tools.

6.3 Modeling and Coding Standards

In many software projects, in particular for safety-critical applications, coding standards have been introduced and have proven successful. Since most programming languages are very rich in expressability, the unrestricted use of all concepts and constructs of a language does not only create a very heterogeneous style of code, but such code is usually much more prone to errors.

Numerous coding standards have been developed over the years, which can be taken "as is", or can be customized. The use of coding standards is mostly enforced during (manual) code review and inspection, unfortunately late in the process. Nowadays, numerous tools, both commercial and open-source, exist, which can automatically check the conformance of the code (fragments) to a given coding standard.

In a model-based environment, the use of "coding" standards is even more important. This is the case not only because multiple representations of information and knowledge exist and are expressed in different formalisms, but also for the safe interaction with automatically generated code, which, per se strictly adheres to coding standards. Here, "coding standard" has to be extended to incorporate *modeling standards* as well as coding standards in the stricter sense.

Our survey [1] revealed that in the vast majority of projects, one or several coding and modeling standards have been used. There is a large number of different coding standards that have been developed and published over the years. In many cases, a coding standard suitable for the project at hand can be found immediately. Otherwise, an existing coding

standard can form the basis for a customized one. When a projects decides to adopt a modeling or coding standard, the following issues should be considered:

- A modeling/coding standard has the main purpose to help and guide the developers. Therefore, the developers should be involved in the decisions related to which coding standard should be used and how it should be customized. Coding standards, which are dictated by management only, and which are not commonly adopted are counter-productive.
- The standards must be fit and suitable for the intended project and purpose.
- A coding and modeling standard *must* be defined and adopted before any models or code is developed. Later changes are extremely costly and make the code harder to debug and maintain, because familiarity with the code might be lost due to re-structuring and (syntactic) code changes.
- Layouting of the model (or code) is very important for its readability. Yet, often it is a very personal issue and individual developers have their own (and strong) preferences. The availability of customizable tools for code and model layouting makes it easier for the developers to use their own style, while a uniform, project-wide layouting style can be generated automatically. The check of correctness for these code transformation can be accomplished easily.
- Coding and modeling standards go hand-in-hand with documentation standards. Adherence to standards enable the use of powerful documentation generation tools for code (e.g., javadoc, doxygen [51]) and models (e.g., Simulink's report generator [9]).
- Consciously developing artifacts according to standards requires some additional effort. If that task is spread out (and monitored) during the entire duration of the project and the effort is honored, long and ugly "documentation backlogs" can be avoided.
- Tools for the automatic conformance checking of code and other artifacts need to be customized *early* in the process. These tools do not require much computational resources and thus should be run often, at least during the nightly build. Here again, reasonable feedback to the developers must be provided by the tools. If a developer is overburdened by mountains of nitty-gritty details, keeping her or him from doing important work, coding and modeling standards will be ignored more and more.
- Coding and modeling standards do not only require buy-in from all the stakeholders but also training/familiarization. If combined with training for MBSwE tools and AGC tools, the additional overhead should be minimal.

6.4 Toward Certification of MBSwE and AGC

With an increasing level of criticality of the software the requirements of *certification* become more stringent. In contrast to V&V, which aims at showing/analyzing that the software does what the requirements say and does so flawlessly, certification is often seen as a process with the aim of demonstrating to a certification authority that due diligence has been done during all phases of the software development and V&V.

Typically, certification of a project is done according to a standard that prescribes which artifacts, plans, and documentation must be produced, and which rigor has to be applied during V&V. For example, several standards (e.g., DO-178B/C [52]) require different testing coverage levels depending on the criticality of the software.

Standards, which are frequently used [1] include DO-178B/C [52] and its European equivalent ED12B (published by EUROCAE), ISO 26262 [53], IEC 61508 [54], or IEEE-1012. For NASA missions, NPR 7150.2 is relevant.

In particular, for higher levels of criticality, standards can require that the software tools used in the project are certified to at least the same level of criticality as the target software (e.g., DO-178B/C). That requirement can carry a substantial burden, since most MBSwE and AGC tools are not certified, or even certifiable because of their complexity. Moreover, some standards (e.g., DO-178B/C) require that the tools to be certified within the actual individual project requirement. This means a tool does not come with a stamped "certified according to" label. Rather certification of the tool has to be carried out by the customer within their project environment. Therefore, some tool vendors sell certification kits which include documentation about the tool and its development process and provide test suites that, when executed at the customer's site, can demonstrate the code coverage, necessary for the required standard.

The application of MBSwE and AGC in a project with a required certification level requires considerable detail and planning. A considerations should include the following:

- The use of MBSwE and AGC in a project with goal for certification must be planned early. Development and V&V plans should be discussed with the certification authorities as early as possible.

- Some (commerical) MBSwE tools already come with certification support with respect to a given standard. This certification support, however, must be configured carefully, as certification usually is project-specific and not tool-specific.

  Even when aquiring a certified tool, resources need to be set aside to carry out the project-specific tool certification and to integrate the certification results into the development and V&V plan.

- Documentation plays an extremely important role in certification. Since many MBSwE tools can generate numerous documentation artifacts and allow the linkage between artifacts (e.g., between model elements and requirements), a lot of effort can be saved if the auto-generated artifacts are incorporated into the certification documents.

- The use of some tools might yield a "verification credit", which, further down the road might reduce V&V efforts. These verification credits must be documented and explained carefully to facilitate their acceptance with the certification authority.

- Experiences and results of past certification efforts are gold-nuggets of knowledge and should be documented and kept carefully.

- Certification is a very human-centric process, so early, frequent, and clear communication with the certification authorities is paramount. That communication should also flow into the certification documents.

Chapter 7

Benefits and Challenges

The use of MBSwE and AGC offers opportunities and holds promise for many benefits, but it also imposes challenges for the software development and SWA teams. In this Chapter, we summarize the benefits and challenges based on the responses to survey questions [1,5], as well as the qualitative and quantitative findings of the two cases studies [2], [3].

7.1 Benefits

Based on the survey respondents, the overall experiences with MBSwE and AGC were positive. Specifically, 74% and 69% (out of 81 and 82 respondents of our survey [1], respectively) rated their overall experiences with MBSwE and AGC as either "good" or "excellent", with a median of "good" in both cases. (Rating scores included "poor", "fair", "satisfactory", "good", and "excellent".)

The respondents of our survey identified the following benefits:

- **MBSwE and AGC increased the productivity.** Vast majority of respondents to our survey [1] either "agreed" or "strongly agreed" that MBSwE and AGC increased the productivity (81% and 87%, respectively), with median values of "agree" in both cases. (Based on 79 respondents.)

- **MBSwE and AGC resulted in better maintainability and quality (i.e., fewer bugs).** 85% and 79% of the respondents (out of 77 and 80 respondents) either "agreed" or "strongly agreed" that MBSwE and AGC resulted in better maintainability and quality (i.e., fewer bugs) of their projects, respectively [1].

- **Getting the model "up and running" and have the production-code generated are not the only benefits of using MBSwE/AGC.** Many respondents of our survey [1] are using MBSwE tools to generate a multitude of artifacts related to documentation, testing, standards, and certification. In addition, models seems to facilitate communication between the members of the team or teams.

The development team of the NASA mission used as our first case study [2] found the **MBSwE and AGC approach based on dictionaries to be much more useful that the traditional MBSwE approach based on state charts.** In particular, the AGC-D files, which were auto-generated using the dictionaries as input, were the least fault prone, compared to AGC-M files (auto-generated from MagicDraw state charts) and handwritten files.

The development team of the NASA mission used as our second case study [3] identified the following benefits:

- **The well defined interfaces and model-based development allowed for easy integration with other software applications** provided by the external development teams.
- **Real Time Services (RTS) library[1] provided very good support for communications, timers and message logging.**
- **Interfacing between Rational Rose RealTime and VxWorks was relatively seamless** and porting the software to newer versions of VxWorks was made easier by the close collaboration and partnership between IBM Rational and Wind River.

7.2 Challenges

Next, we summarize the challenges posed by the use of MBSwE and AGC approaches.

- **The transition from a traditional software development process to MBSwE/AGC does not appear to be easy.** Specifically, 58% of the 75 survey respondents either "strongly disagreed" or "disagreed" that the transition was easy, with a median value "disagree" [1].
- **There seems to be a steep learning curve on how to use the MBSwE approach and tools effectively.** 61% of 78 survey respondents either "strongly disagreed" or "disagreed" that learning the effective use of MBSwE/AGC was easy, with a median value "disagree" [1]. Anticipating the need for training, the development team of the NASA mission used as our second case study [3] took three months for learning the MBSwE approach and UML, and used the Rational Rose Development Suite to produce prototypes. According to developers, it appeared that some team members needed more time to adjust to the MBSwE and Object Oriented (OO) programming paradigms.
- **Lack of knowledge and familiarity with MBSwE/AGC can lead to fear and concerns regarding its use.** Specifically, 66% of 72 respondents either "disagreed" or "strongly disagreed" that the lack of knowledge did not lead to fear and concerns, with a median value "agree" [1].
- **Extra effort is needed to develop the models, which may be overlooked or underestimated** [2]. Due to this extra effort, the generation of the first code might take some time.
- **It is an open question when to stop including details in the model** [2]. Beyond some level of details, it appears that no one other than the developer who has built the model could understand the model.

[1]RTS library [55] is a run-time framework used by the target code that is generated from a UML real-time model in Rational Rose RealTime. RTS isolates the application code from the target environment, so that the same real-time application can be built for multiple target environments. In addition, the RTS library also provides services (such as communication, timing, dynamic structure, concurrency, and message based processing), which the application can use at run-time.

- **Models tend to obscure the lower level details present in the AGC**, according to developers of the mission used as our first case study [2].
- **AGC typically in not easily readable** because it is not intuitive and lacks comments [2]. Based in developers' experience, the readability of AGC is typically improved once one knows and understands the patterns used by the tool (e.g., MagicDraw patterns in the case of one of our case studies [2]).

Chapter 8

Summary & Recommendations

Obviously, MBSwE and AGC have established role in design, development, and deployment of safety-critical systems in many industrial areas. As discussed in Chapter 7 the use of MBSwE and AGC leads to benefits and brings improvements. However, there are also many issues with MBSwE and AGC that need to be carefully considered throughout the entire project life (and beyond) and there does not seem to be a unique and/or easy way to handle them. In the following, we list recommendations that are supported by the survey results and case studies based on NASA missions.

- **Early and detailed planning of the MBSwE endeavor and its implications throughout the development, V&V, and maintenance phases are instrumental** in order to be able to use all benefits of such approaches.
- **MBSwE and AGC tool(s) or tool chains need to be selected carefully; not all tools work equally well with the selected process(es) and standards.** Often, combinations of tools seem to help solve problems in MBSwE and AGC.
- **Both open-source tools and commercial tools are actively being used in MBSwE projects and there does not seem to be a clear preference toward one or the other.** However, only a few tools/tool chains are certified for a specific standard (e.g., DO-178B/C). Nevertheless, many respondents to our survey [1] reported successful use of MBSwE and AGC with non-certified tools.
- **There is a necessity of knowledge and experience in model development,** which should not be underestimated, in particular for larger projects, where multiple teams need to collaborate in the modeling and design process. MBSwE does not come for "free" (even using open-source tools). Many respondents to our survey [1] and NASA mission developers [3] mentioned the steep learning curve and the necessity of training, which needs to be planned in advance.
- **For each new project using MBSwE and AGC, time and resources need to be set aside for (1) setting up the model-based environment/tools, (2) training the team on MBSwE and "get them to accept it"** [1], [3]. These activities should be done as early as possible, in particular *prior* to requirements phases, as MBSwE methods provide many advantages during this phase.
- Due to this ramp up and modeling efforts, **the generation of first code might take some time, so team members should not "loose their patience".**

- **Model-based phase should not be over with the deployment.** In the majority of projects, the models were maintained after deployment phase [1]. Keeping models maintained and current during the entire life-cycle of the product (especially after deployment) and even after end of the project life time can bring substantial benefits, such as (1) easier maintenance and knowledge transfer to other teams, (2) facilitated model reuse.

  Again, careful and early planning is essential (e.g., extended tool licenses, lifetime of tool versions, etc.) Changes in tool versions during the project can lead to bugs and might require time-consuming and often unnecessary modifications to the models. **In many cases, it is advisable to select and then freeze the versions of the tool chains, even if the latest tool features might not be available.**

- **Concise modeling standards, like coding standards, facilitate, by enforcing a uniform structure of the models, the readability of the models by other team members / teams, and often reduce the number of modeling errors.** Modeling and coding standards have been used by many projects to their advantage. Many tools provide some form of automatic checking for modeling standards.

  The early definition and adoption (by everyone!) of a modeling standard is essential. A number of well-accepted and available standards exist.

- **Although less important for AGC, the definition of a coding standard throughout the entire project, i.e., for auto-generated code as well as for handwritten code can improve readability and helps to avoid coding errors and time-consuming nuisances.** If possible, the code generator should be customized to adhere to the selected coding standard. That might take some effort early on, but usually will pay off.

- **The models and AGC should be synchronized often** as they tend to diverge over time. Software practices like the "daily build" can, and should, be extended within the MBSwE frameworks and tool chains.

- **Once, code has been generated, it should be left alone and not be modified manually.** Manual modification, even if done using scripts, can be significant source of bugs and cause multiple problems when the models or the tools change.

- **MBSwE does not mean you only have to do V&V on the model level.** Our survey [1] and case studies [2], [3] showed that bugs were found in the model and in the auto-generated code. Thus, V&V activities are necessary on both levels, where they can be tailored toward model or code analysis. Bugs related to requirements and design, architecture, documentation, or structural logic often manifest themselves already on the model level; structural and data (initialization) related bugs tend to show more often on the code level. Again, bug-catching as early as possible and on the higher model level should be preferred. Also note that, as discussed in Section 7.2, debugging of auto-generated code can be extremely difficult and time-consuming because AGC is often hard to read.

- **Certification standards such as RTCA DO-178B/C, ISO 26262, ISO 61508 and their European equivalents such as the EUROCAE ED12B, appear not to be very popular or highly utilized among the spectrum of the survey**

respondents [1]. It should be noted that not all of these standards are specifically tailored toward the use of MBSwE and AGC. Therefore, the instantiation of certification standards in a project using MBSwE and AGC is not only difficult and time-consuming, but often this knowledge (including tool customizations) are not sufficiently carried over to other projects.

Note that the recommendations listed in this Chapter were compiled based on the lessons learned from the research work on our SARP funded project. This list was not supposed to be exhaustive. Rather, our goal was to concisely document the recommendations based on our findings so NASA missions that use or plan to use MBSwE and AGC can benefit from them and subsequently augment the list based on their findings and experiences.

Bibliography

1. Goseva-Popstojanova, K.; Kahsai, T.; Knudson, M.; Kyanko, T.; Nkwocha, N.; and Schumann, J.: Survey on Model-Based Software Engineering and Auto-Generated Code. , NASA Software Assurance Research Program (SARP) Technical Report, Oct. 2016.

2. Goseva-Popstojanova, K.; Kyanko, T.; and Nkwocha, N.: Model-Based Software Engineering and Auto-Generated Code at NASA: A Case Study based on an Ongoing Mission. , NASA Software Assurance Research Program (SARP) Technical Report, Nov. 2017.

3. Goseva-Popstojanova, K.; Kyanko, T.; and Nkwocha, N.: Model-Based Software Engineering and Auto-Generated Code at NASA: A Case Study based on a Mission under Development. , NASA Software Assurance Research Program (SARP) Technical Report, Sept. 2018.

4. Schumann, J.; Kahsai, T.; and Knuds, M.: D2A: Report on V&V Architecture for Model-Based Software Engineering. , NASA Software Assurance Research Program (SARP) Technical Report, 2016.

5. Schumann, J.; and Denney, E.: Customer Survey on Code Generators in Safety-Critical Applications. ESAS 6G-PS 1.1.2.3 Report, NASA, 2006. URL `http://ntrs.nasa.gov/archive/nasa/casi.ntrs.nasa.gov/20060051796.pdf`.

6. Boehm, B.: A Spiral Model of Software Development and Enhancement. *SIGSOFT Softw. Eng. Notes*, vol. 11, no. 4, Aug. 1986, pp. 14–24. URL `http://doi.acm.org/10.1145/12944.12948`.

7. Boehm, B.: Spiral Development: Experience, Principles,and Refinements. 2000-SR-008, CMU-SEI, 2000.

8. Rumbaugh, J.; Jacobson, I.; and Booch, G.: *Unified Modeling Language Reference Manual, The (2Nd Edition)*. Pearson Higher Education, 2004.

9. The Mathworks Company: Matlab, Simulink, Stateflow. URL `mathworks.com`.

10. ISIS: Generic Modeling Environment (GME). URL `http://www.isis.vanderbilt.edu/projects/gme/`.

11. IBM: Rational Dynamic Object Oriented Requirements System (DOORS). 2018. URL `https://www.ibm.com/us-en/marketplace/rational-doors`.

12. RTCA: DO-178C/ED-12C: Software Considerations in Airborne Systems and Equipment Certification. 2012. URL http://www.rtca.org.

13. Open Ameos: Open Ameos Modeling Environment. URL https://www.scopeforge.de/cb/project/8.

14. Ameos Users Group: A Quick Tour of OpenAmeos. URL https://www.scopeforge.de/cb/displayDocument/UMLEVAL.pdf?doc_id=2386.

15. Dassault group: Autosarbuilder. URL http://www.3ds.com/products-services/catia/products/autosarbuilder/.

16. Autosar: URL https://www.autosar.org.

17. Bridgepoint. URL https://xtuml.org/about/.

18. One Fact, Inc: Bridgepoint Pro. URL http://onefact.net/products/bridgepoint-pro/.

19. ISIS: DREMS: Distributed Real-Time Managed Systems. URL http://www.isis.vanderbilt.edu/drems.

20. dSPACE GmbH: URL www.dspace.com.

21. Eclipse.org: Papyrus modeling environment. URL https://eclipse.org/papyrus/.

22. SysML.org: SysML Open Source Specification Project. URL http://sysml.org.

23. Eclipse.org: Xtend. URL https://marketplace.eclipse.org/content/eclipse-xtend.

24. Eclipse.org: XText. URL http://www.eclipse.org/Xtext/.

25. Sparxsystems.com: Enterprise Architect. URL http://www.sparxsystems.com/products/ea/.

26. Abstract Solutions: iUML. URL http://www.abstractsolutions.co.uk/PRODUCTS/iuml/.

27. No Magic, Inc: MagicDraw. URL http://www.nomagic.com/products/magicdraw.html.

28. National Instruments: NI MATRIXx. URL http://sine.ni.com/nips/cds/view/p/lang/en/nid/12153.

29. Mbeddr: Mbeddr tool. URL http://mbeddr.com.

30. The Modelica Association: Modelica. URL www.modelica.org.

31. Wikipedia: Modelica. URL https://en.wikipedia.org/wiki/Modelica.

32. Siemens: LMS Imageine Lab Amesim. URL https://www.plm.automation.siemens.com/en_us/products/lms/imagine-lab/amesim/.

33. Dassault Systems: CATIA. URL http://www.3ds.com/products-services/catia/capabilities/.

34. Dassault Systems: Dymola. URL http://www.3ds.com/products-services/catia/products/dymola/key-advantages/.

35. JModelica: URL http://www.jmodelica.org.

36. OpenModelica: URL www.openmodelica.org.

37. EaSE Group: Merapi Modeling. URL https://www.ostfalia.de/cms/de/ivs/ease/Merapi/MerapiModeling.html.

38. ObjecTime Developer. URL https://en.wikipedia.org/wiki/ObjecTime_Developer.

39. Quantum Leaps: QP frameworks. URL http://www.state-machine.com/products/index.html#QP.

40. IBM: Rational Rose Modeler. URL http://www-03.ibm.com/software/products/en/rosemod.

41. IBM: Rational Rhapsody family. URL www.ibm.com/software/awdtools/rhapsody/.

42. Wikipedia: Rational Rhapsody. URL https://en.wikipedia.org/wiki/Rational_Rhapsody.

43. ROSlab. URL http://precise.github.io/ROSLab/.

44. Esterel-Technologies: Scade Suite. URL esterel-technologies.com.

45. UPPAAL. URL http://www.uppaal.org.

46. Visual Paradigm: Effective IT System Design, with UML & ERD. URL https://www.visual-paradigm.com/features/.

47. Microsystems, S.: rpcgen Programming Guide. 2018. URL https://docs.freebsd.org/44doc/psd/22.rpcgen/paper.pdf.

48. Siemens: Polarion Requirements. 2018. URL https://polarion.plm.automation.siemens.com/products/polarion-requirements.

49. Marcil, L.: Model-based design amp; code generation: A cost-effective way to speed up HMI certification. *2011 IEEE/AIAA 30th Digital Avionics Systems Conference*, Oct 2011, pp. 1–27.

50. Hamill, M.; and Goseva-Popstojanova, K.: Exploring the missing link: An empirical study of software fixes. *Software Testing, Verification and Reliability Journal*, vol. 24, no. 8, 2014, pp. 684–705.

51. doxygen cross-platform documentation system. URL http://www.doxygen.org.

52. RTCA: DO-178B: Software Considerations in Airborne Systems and Equipment Certification. 1992. URL http://www.rtca.org.

53. ISO 26262 Road Vehicles functional safety. 2011. URL https://www.iso.org/standard/43464.html.

54. IEC 61508. URL http://www.iec.ch/functionalsafety/.

55. The RT Services Library. https://www.ibm.com/developerworks/community/wikis/home?lang=en#!/wiki/b7da455c-5c51-4706-91c9-dcca9923c303/page/325220ca-b17d-4ea6-b382-4c704dbad0af/version/13f44627-fd96-4341-91fb-611a01b3c3fb. Accessed: 2018-06-01.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704–0188

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
01-06-2018	Technical Memorandum	01/2016–05/2016

4. TITLE AND SUBTITLE
Guidebook on Model-Based Software Engineering and Auto-Generated Code

5a. CONTRACT NUMBER
NNA14AA60C/NNG12SA03C

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER
SARP

6. AUTHOR(S)
Katerina Goseva-Popstojanova, Johann Schumann, Noble Nkwocha, Matt Knudson

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
NASA Ames Research Center, Moffett Field, CA 94035
West Virginia University, Morgantown, WV 26506
NASA IV&V Facility, Fairmont, WV 26554

8. PERFORMING ORGANIZATION REPORT NUMBER
L–

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
National Aeronautics and Space Administration
Washington, DC 20546-0001

10. SPONSOR/MONITOR'S ACRONYM(S)
NASA

11. SPONSOR/MONITOR'S REPORT NUMBER(S)
NASA/TM–2018–TBD

12. DISTRIBUTION/AVAILABILITY STATEMENT
Unclassified-Unlimited
Subject Category 61
Availability: NASA STI Program (757) 864-9658

13. SUPPLEMENTARY NOTES
An electronic version can be found at http://ntrs.nasa.gov.

14. ABSTRACT

Model-based Software Engineering (MBSwE) is a powerful paradigm to develop mission- and safety-critical code for NASA missions. Despite obvious advantages and numerous tools that are to support MBSwE, many intricacies and potential issues can endanger the budget and schedule of the software project and carry the risk of producing code of low quality and reliability. This guidebook is focused on MBSwE and Auto-generated Code (AGC) with a goal to provide a comprehensive overview of the field and popular tools, make the reader aware of important issues, provide practical guidelines for selection of suitable processes and tools, describe the modeling-standards, how to deal with synchronization and maintenance issues, and how to set up a powerful, tool-supported verification and validation process. This guidebook was developed in support of the NASA Software Assurance Research Program (SARP) and draws from a NASA and industry-wide survey on this topic, and case studies based on two NASA missions.

15. SUBJECT TERMS
software engineering, model-based software development, code generation

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			STI Information Desk (email: help@sti.nasa.gov)
U	U	U	UU		19b. TELEPHONE NUMBER *(Include area code)* (757) 864-9658