Experiences with Model and Autocode Reviews in Model-based Software Development

Ingo Stürmer
Member of the ACM
stuermer@acm.org

Mirko Conrad, Ines Fey, Heiko Dörr
DaimlerChrysler AG
Research and Technology
{mirko.conrad | ines.fey | heiko.doerr} @daimlerchrysler.com

ABSTRACT
Through the introduction of model-based development, paradigm models became first class citizens in the development of in-vehicle software and are thus also object to strict quality assurance. Just as code reviews are widespread in classical software development, models also have to undergo a stringent review procedure – particularly if they serve as a basis for automatic software implementation by means of model-based code generators. In addition to model reviews, the generated production code (autocode) must be reviewed by performing so-called autocode reviews. This paper presents our procedure for a combined model and autocode review and provides examples, lessons learned, as well as significant experimental results drawn from a typical automotive embedded software development project.

Categories and Subject Descriptors
D.2 [Software Engineering]: Software/Program Verification, Testing and Debugging – Code inspections and walk-throughs, Metrics – Complexity measures

General Terms
Design, Reliability, Human Factors, Languages, Verification.

Keywords
Model-based development, Simulink, Stateflow, automatic code generation, modeling guidelines, autocode review, model review, complexity

1. INTRODUCTION
Model-based development is a way in which numerous automobile suppliers and manufacturers are responding to increased demands on the development of embedded software [2]. The model-based approach is characterized by the seamless use of executable graphical models for specification, design and implementation, using commercial modeling and simulation tools such as MATLAB/Simulink/Stateflow [16] or ASCET-SD [15]. These tools use data-flow oriented block diagrams and extended state machines as modeling paradigms. In contrast to manually created code these models can be early simulated and validated in their respective development environment. For that reason, such models are termed executable specifications. In model-based development, the function to be developed is created in evolutionary steps and thus occurs in different representational forms which are based on each other. During this model evolution, a physical model (PM) is created that represents the control function to be developed in an implementation-independent way and describes its behavior, for example, regarding external stimuli such as continuous input signals or events. The PM typically computes the algorithms using floating-point (FLP) arithmetic. However, for production code generation, the PM has to be revised - for example, function parts are distributed to tasks, and augmented with necessary implementation details. Furthermore, the FLP arithmetic is adjusted to the fixed-point (FXP) arithmetic of the embedded processor. This adaptation results in an implementation model (IM), which contains complete information necessary for production code generation by means of a code generator (CG).

The increasing complexity of embedded controller functionality also leads to a higher complexity of the IMs used for automatic controller code generation (autocoding). Modeling tools, which are typically used for IM design, cannot sufficiently support the developer in dealing with increasing model complexity. For example, one and the same issue can be modeled in different ways, which intensifies problems such as coping with functional design, limited resources available, and model/code maintenance. It is possible to deal with these problems by means of constructive methods (e.g. modeling guidelines) or analytical ones (e.g. reviews). Reviews on code level are an integral part of traditional software development. In model-based development, however, autocoding changes the aims and emphasis of a code review. Functional as well as structural-related review aspects can be shifted up to model-level. For that reason, model reviews are becoming increasingly important.

This paper reports on the successful application of combined model and autocode reviews in an embedded automotive software development project. Furthermore, we present and discuss our results and the lessons learned.

The remainder of this paper is structured as follows: Section 2 illustrates the need for and the aims of model and autocode reviews by means of an example. Section 3 presents the realization of our review approach and the development artifacts to be reviewed. The results obtained and lessons learned are summarized in Section 4. Section 5 concludes the paper.

2. REVIEWS IN MODEL-BASED DEVELOPMENT
In traditional software development, code reviews are an inherent part of analytical quality assurance (QA). But the goal of code reviews changes when autocoding is applied, as it is done in the
case of model-based development of in-vehicle control software. Aspects such as checking optimizations performed by CGs are increasingly coming to the fore [5]. Since IMs serve as a blueprint for autocoding, some, but not all, review aspects should be raised to the model level. For instance, it is easier to check functional or structure-related aspects on model level whereas the interfaces between manually and automatically coded software parts are more efficiently reviewed on code level.

2.1 Model Reviews

Executable models which were created early in model evolution can be regarded as executable specifications. They reflect the functional requirements of the control function to be developed in a constructive manner. In order to review those models, review procedures for requirements specifications, e.g. Fagan inspections (cf. [14]), can be adapted. For models which already contain implementation details, additional issues have to be taken into account. The aims of such model reviews are:

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

Model reviews are often guided by an in-house set of modeling and review guidelines, which are commonly realized as a check list cf. [8]. During the model review, a series of findings with suggestions and comments on individual model parts are gathered and recorded with a reference to the affected model elements. The references to the model elements enable the developer to track which parts of the model may have to be revised.

Example 1

The following two realizations of a buzzer are exemplifying the need for model reviews. The first realization (Figure 1) was created at an early development stage of our project and does not conform to our modeling guidelines. Furthermore, it represents a model that contains several problems that are not obvious at the first glance (esp. for Flowchart beginners). The two possible and semantically comparable realizations of a buzzer are presented in Figure 1 (Flowchart, variant A) and Figure 2 (Stateflow, variant B). The generation of an acoustic warning (beep) is controlled by the flag BEEP_flag_buzz. When initially activated, the flag for the activation of the beep BEEP_flag_buzz is turned to true (1) during t_CON. After t_CON the flag is turned back to false (0). The buzzer beeps three times. In variant A, which has been created at an early development stage of our project, the buzzer is realized as a Flowchart diagram [16] that typically consists of transitions and connective junctions. A connective junction defines a decision point between possible paths of a transition, whereas a transition can bear a complex label for checking specific conditions (e.g. runs > 0) and for performing actions (e.g. counter++;). This modeling paradigm is comparable to if-then-else control structures used in e.g. C. The appropriate use of Flowcharts for automatic controller code implementation often produces efficient C code with small overhead. Modeling with Flowcharts, however, allows a quasi free C code programming. The assignments in the curly braces, for example, can contain an arbitrary number of C statements. As a consequence, classical programming errors can occur, such as the faulty use of a bitwise AND (&) instead of the logical one (&&). Such human-related sources of error are often neglected or underestimated in the literature e.g. [3], [6]. Inappropriate modeling must be considered as an additional error source. For example, timing behavior, such as the fractions of time the buzzer is activated or disabled, is realized by incrementing the local counter variable. This kind of modeling temporal behavior is often used by Stateflow beginners. However, when using the model for production code generation, such modeling of temporal behavior implies intrinsic errors. Timing is no longer independent of the processor’s execution speed or time for computations assigned by the operating system respectively. In variant A, the flowchart realization of the buzzer is more technical, rather than intuitive, since the functionality of the buzzer is, in principle, state-based. After all, the expected efficiency improvements
on code level are largely compensated by the disadvantages stated above.

The realization of the buzzer as a Stateflow state machine (Figure 2) is easier to understand intuitively. In variant B, temporal behavior is modeled by using Stateflow’s after operator, which allows a precise definition of how long a state is activated - for example, when the transition from state On to Off is executed. Timing is “incremented” via the use of the Stateflow CLK event. In doing so, timing is independent of the processors’ execution speed or resource allocations of the operating system. In contrast to variant A, this variant is easier to parameterize (e.g. for double beeps). In summary, it can be stated that, on the one hand, modeling by using states and events increases understandability and readability of the model, while, on the other hand, the use of states and events is problematic in projects where resource consumption (e.g. required RAM/ROM size) is of decisive importance, since both produce a large overhead. For that reason, a compromise between understandable modeling and code efficiency must be found. Modeling guidelines must clarify advantages and disadvantages of different modeling techniques and patterns.

2.2 Autocode Reviews

Compared to code reviews in traditional software development, autocode reviews can be seen more as a combination of model and code reviews rather than as being solely code reviews. This means that many review tasks which are performed at the code level during a traditional software development process can be shifted towards the model level. But since current CGs are not yet trusted enough to allow their use without reviewing the generated code [5], autocode reviews are still a good software engineering practice. The aims of autocode reviews in such a context are:

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

Example 2

The following C code fragment was generated from a Flowchart:

10: if (flag==2) {
11:     if (flag==2) {
12:         NO_SIGNAL=0;
13:         SIGNAL_L1=1;
14:         SIGNAL_L1_en();
15:     } else {
16:         NO_SIGNAL=0;
17:         SIGNAL_L1=1;
18:         SIGNAL_L1_en();
19:     }
20: }

As can be seen, both if-statements in lines 10 and 11 contain the same condition. For that reason, the else-branch (lines 15-19) of the second if-statement can never be executed and therefore contains dead code (bold). The identical if-statements in line 10 and 11 were caused by inappropriate Flowchart modeling. The dead code was detected during an autocode review which was performed after the model and the autocode had already been tested and also been checked by means of a model review. This is a typical example of a faulty model construct which is easier to detect in the autocode rather than in the model. Furthermore, this example shows that dead code can obviously be generated, which is often excluded per se when using a CG cf. [10], [11].

3. MODEL AND AUTOCODE REVIEWS IN PRACTICE

This chapter reports the authors’ experiences with model and autocode reviews in a large-scale development project. The first part provides an overview of the artifacts to be reviewed in terms of size and complexity as well as the tools used. The second part describes the review approach and its integration into the overall model-based QA concept.

3.1 Development Artifacts to be reviewed

3.1.1 Implementation Model

First, a PM was created with MATLAB R12.1 using both Simulink and Stateflow modeling languages. Simulink and Stateflow models are executed within the MATLAB environment as time-continuous systems using FLP arithmetic. In order to have an autocode-ready IM, the PM has to be transformed into a time-discrete model using FXP arithmetic. We used the TargetLink [4] CG for production code generation. In order to do so, the Simulink and Stateflow portions of the PM had to be converted into a TargetLink model. During this conversion the Simulink blocks are automatically replaced by replicas from the TargetLink block library. TargetLink uses the original Stateflow diagrams for code generation; there are no corresponding TargetLink blocks. The reason for this conversion is that TargetLink needs additional information (besides discrete time-behavior) for code generation, such as scaling parameters and data types. The TargetLink model, which resulted from this conversion, represents the IM of the control algorithm. The IM is structured and designed in compliance with our in-house set of modeling guidelines cf. [8]. These guidelines propose specific modeling patterns and CG configurations with the purpose of generating safe and efficient autocode.

The IM metrics are presented in the first section of table 1 (see last page) with detailed information about 8 of the major top level subsystems (M1-M8). The IM consists of ~9,000 blocks, which where partitioned into ~900 subsystems with a maximum subsystem depth of 14 levels. Note that not all blocks are necessary for code generation such as scopes, memory read/write blocks etc. However, in order to get a notion of the model’s complexity, we measured the cyclomatic complexity of the IM. The cyclomatic complexity measure CYCMod is a measure of the structural complexity of a model which approximates the McCabe complexity measure for code generated from it. CYCMod is defined as follows:

\[
CYC_{Mod} = \sum_{i=1}^{N} (o_i - 1)
\]

\(N\) is the number of decision points which the model or subsystem contains. \(o_i\) is the number of outcomes for the \(n\)-th decision point. The complexity number computed is incremented by 1 for atomic
The model size and complexity values were computed by an in-house tool and the Simulink Verification and Validation toolbox [16] respectively. The correlation between metrics on model and on code level are discussed in [21].

3.1.2 Autocode
For production code generation, we used the TargetLink 1.3p2 CG [4] applying standard CG optimizations (e.g. constant folding, dead code elimination, interblock optimizations). But we did not use the target optimization module, which consists of library code patterns that are particularly efficient for specific compiler/processor combinations cf. [7].

The autocode metrics are presented in the first section of table 2. This section shows sizes and complexity measures for 8 autocode modules corresponding to the major subsystems considered in table 1 as well as for the entire autocode. Each function module is generated as a separate file. In order to gain an impression on the optimizations performed by the CG, complexity measures for the optimized as well as for the unoptimized code are provided. The optimized C code, for example, consists of ~10,000 executable lines of code (XLN) organized into a total of 49 functions. The unoptimized C code consists of ~15,000 XLN organized into a total of 54 functions. Size and complexity measures for the autocode were computed with the QA-C tool [18]. Note that the total XLN comprises the XLN of the entire autocode, not only of the 8 autocode modules presented in table 1.

3.2 Our Review Approach
The different reviews have to be carefully integrated into an overall QA approach for model-based development, which also incorporates methods such as testing and static analysis. In the development project under consideration, the quality assurance activities were performed in the following order, taking into account previous experiences and the expected efforts of the reviews cf. [1]:

1. model reviews
2. IM tests (FLP modus)
3. static analyses of the autocode
4. autocode reviews
5. IM regression tests (FLP modus)
6. back-to-back tests IM vs. FXP autocode

Code generation was performed between steps 2 and 3. Each of the steps 1-6 was followed by an evaluation phase. At the end of each phase, detected deficiencies, general findings, and possible model modifications were assessed with the aim of correcting specific problems and a re-generation of the autocode.

During the different reviews, a series of findings with suggestions and comments on individual model parts was gathered. The findings are divided into 5 classes (questions, editorial remarks, un-critical remarks, important remarks, and critical remarks). All findings were recorded in DOORS [12] and each comment had to reference the affected model/autocode element. Thereby, it is possible to use the DOORS mechanisms for processing and managing deficiencies and findings.

3.2.1 Model Review
The IM reviews took place as focused model reviews according to a checklist with 10 review questions (top 10 model review questions). The main focus of the top 10 questions was not to check general modeling issues, but to ensure safe and efficient code generation. Each module was reviewed by 2 or more persons according to the “4-eye-principle”.

Referencing model elements with DOORS means that it is possible to track which parts of the model may have to be reworked. In order to make the discussion and processing of the review comments as efficient as possible, precise assignment of the individual review comments to the model elements concerned is needed, at least on subsystem level or state machine level. For this reason, a specific instance of the ToolNet framework [13] was used for conducting the model review. ToolNet, a service-based integration framework, supports the management of comments, the links to model objects as well as the realization status, thereby facilitating the integrated usage of Simulink/Stateflow and DOORS in our application context. Furthermore, finding annotated model elements is also a long and expensive process. In the review process, the comments are discussed sequentially and the action that needs to be taken is identified during inspection meetings. This actually calls for direct access to the affected model element. Here automatic navigation from the comments to the annotated model element saves a great deal of time.

3.2.2 Autocode Review
The autocode review followed the two-phase autocode review procedure proposed by the authors cf. [1]. Each phase is guided by its own checklist consisting of questions and examples. The first phase concentrates on aspects such as comprehensibility, structure, and form of the code. This phase basically clarifies whether the requirements for an effective execution of the second review phase have been fulfilled. In general, the first phase can be executed in a relatively short time. The second phase is considerably more complex and deals with specific error causes (e.g. incorrect scaling or classic programming errors within the custom code portions). In addition, each code line needs to be mapped back to the respective model pattern and vice versa. A reason for this is that the CG may omit code generation for specific model parts. For example, model parts are modeled in a faulty way and are therefore recognized as “dead paths” by the CG. Furthermore, the CG may generate additional code, for reasons of inappropriate modeling. In doing so, we colored all model parts (e.g. blocks) that can be mapped to the code. Upon completion, every non-colored model part needs to be checked. However, the two-phase separation is largely based on the fact that a single phase emphasizing both of these areas would require a checklist too broad to be feasibly applied.

Findings are recorded during the autocode review in DOORS with a reference to the related code line(s). During the reviews, no editor was available that could facilitate the generation of ToolNet links to specific code lines. Meanwhile, however, this feature is under construction.

4. EXPERIENCES / LESSONS LEARNED
This section shows quantitative data gained from the conducted model and autocode reviews, and reports our experiences and lessons learned.
4.1 Review Effort and Effectiveness

4.1.1 Model Review
Table 1 (section 2) shows the overall and mean review durations for the IM and its major top-level subsystems. Altogether, 236 findings were identified, resulting in 146 revisions of the IM (see Table 1, section 3).

Due to the limited scope of the review, a relatively high number of blocks could be reviewed per hour. The number of reviewed blocks per hour varies significantly between the different subsystems (30…3000 blocks/hr). Smaller subsystems tend to result in disproportionately long review durations, while subsystems with repeated or standardized patterns tend to result in disproportionately short review times. In any case, the number of findings per hour varies much less; between 3.9 and 13.4 findings per hour were identified.

A lesson learned was that the top 10 model review question list certainly helped to focus the review time and to enable short review durations but did not ensure a sufficient review depth. Moreover, the adoption of appropriate tools can lead to a substantial gain in review efficiency. By using the specific ToolNet instance for conducting model reviews (cf. section 3.2) the required effort for conducting model reviews was reduced by 40% compared to an MS Office-based solution.

4.1.2 Autocode Review
The results of the autocode review are presented in the second section of table 2. Altogether, 189 findings were identified, resulting in 20 revisions of the IM (see table 2, section 3).

Autocode needs different review approaches than hand-written code. The two-stage review procedure for autocode proposed by the authors proved successful in the project under consideration, as it made it possible to review between 84 and 257 executable lines of C code per hour. This is higher than the commonly used rule of thumb of 1 XLN per min.

The autocode review duration significantly differs from the models’ one. Reasons for this are: first, the scope of the autocode review was broader and; second, highly optimized FXP code is harder to understand, which makes a mapping back to the related model part even harder.

4.2 Typical Findings

4.2.1 Model Review
Due to the limited scope of the review, the variety of findings gathered was not very high. Typical findings comprise missing or improper block scaling, inappropriate handling of timing, and deficient data type conversions. A different problem is error management: exceptions such as div/0, overflow, or underflow are often recognized by means of configuring the CG (e.g. generation of additional code for protection purposes). For example, a div/0 might occur. A typical solution for this exception is to generate additional code to “protect” the operation. As a result, the division produces the maximal or minimal value of the related data type (e.g. –32768 and 32767 for a signed 16-bit integer respectively), depending on the sign of the divisor. Now consider a multiplication with a value greater than one following this computation. This often results in an overflow. On an embedded processor, this might also cause a processor stall. For that reason, arithmetical exceptions require a dedicated error-management, which must be modeled in order to guarantee correct computation in the subsequent arithmetical operations.

However, in the authors’ opinion, most of the critical and important remarks could have been identified automatically if powerful and adaptable rule checkers for models had been available.

4.2.2 Autocode Review
In contrast to manually created code, autocode was found to have considerably reduced syntactic and data flow errors. We could confirm observations by other studies [22] that model-based generated C code contains fewer errors than hand-coded C by the factors between 2 and 14 relative to average programmers and a factor of 40 compared with the poorest programmer (modeler). However, typical review findings of the autocode review are: (1) improper scaling and definition of data types; (2) model parts are not translated, since they were modeled in a faulty way; (3) code inefficiencies resulting from inappropriate modeling, and (4) CG errors, for example fault translation of arithmetics within Stateflow portions or optimisation errors. Note that there are specific problems that can only be detected within the autocode (e.g. dead or inefficient code). There are static analysis tools available for checking for e.g. dead code, but up to the time we performed autocode review no tool was capable of performing appropriate static analysis on TargetLink code. In the authors’ opinion it is questionable whether an autocode review can be omitted as long as the CG is not being rigorously checked e.g. by means of a generally accepted test suite cf. [9]. This is also the case when using trusted autocode tools, such as SCADe, used in the avionics sector, which perform only a limited amount of optimizations and which can also generate incorrect code due to inappropriate modeling (see [5] for details).

4.3 Model and Autocode Complexity
Autocode that results from a well-designed IM is typically highly optimized. As already stated, the optimized autocode parts are difficult to be mapped back to the original model patterns. For that reason, additional code comments are often generated by the CG. These comments are often supplemented with references to the optimized model parts. This justifies the disproportionately high amount of LINs compared to the XLNs. More precisely, the percentage in reduction of the XLNs is different from the percentage in reduction of the LINs when comparing the unoptimized and optimized autocode parts. Although highly optimized FXP autocode is hard to read, additional comments support the autocode review.

Some strict programming guidelines suggest an upper limit of 10 for a function’s cyclomatic complexity, cf. [19]. The latter is defined as the calculated number of decisions in the control flow of a function. As can be seen in table 2, the cyclomatic complexity CYCAC of the optimized autocode modules ranges between 32 and 348 and is therefore significantly higher than the suggested upper limit. However, this is acceptable from the authors’ point of view since the optimizing of CGs tends to merge model parts and autocode patterns into a few efficient code lines. This justifies the huge values of CYCAC.

The manual of the Simulink V&V toolbox [16] claims the cyclomatic model complexity CYCMOD approximates the McCabe complexity for autocode CYCAC. Furthermore, it states that CYCAC should be slightly higher than CYCMOD because of error checks which are prevalent in the code but not in the model and therefore do not contribute to model complexity. In practice, we could not
confirm this with the TargetLink CG either for the optimized autocode or for the unoptimized autocode. For more than 50% of the modules (M1, M3, M4, M5, M8), the optimized code had smaller CYC values than the model. This means code generation with optimizations can reduce the cyclomatic complexity of the code considerably. Further research has to show whether cyclomatic complexity comparisons between model and code or between code generated by different autocoders from the same model can be used as indicators for the optimization level of the CGs.

Further, we could observe that the difference in XLNs between unoptimized and optimized model parts decreases significantly whenever the model contains huge portions of Stateflow parts (compare modules M1, M2, M3, M5, M8 of table 1 and 2). One may conclude that, for example, the use of huge and nested Stateflow portions can reduce the optimization potentials of the CG. But it does not mean that it is advisable not to benefit from the Stateflow modeling paradigm. Rather, it is the compromise between sophisticated and powerful modeling paradigms, on the one hand, and efficient autocode, on the other hand, that constitutes model-based code generation.

4.4 Combining Reviews and Testing
Reviews as well as test activities have to be closely integrated within the entire quality assurance. The appropriate order in which reviews and other quality assurance measures, in particular testing, should be performed depends on the type of review (focused or full), as well as on the degree of automation of the other QA activities. The far-reaching automation of model-based regression and back-to-back tests allows a careful utilization of the human resources required for reviews. Furthermore, our project has successfully proved that regression testing ensures that no unintended errors are introduced by follow-up review modifications.

In our experience, the application of static analysis tools complements model and autocode reviews in a sensible manner (e.g. checking modeling guideline conformity, language conformance such as ISO/ANSI C, or specific guidelines such as MISRA). Especially improved tool support for static model checks could increase model quality and model review efficiency.

5. CONCLUSIONS
Model-based code generation significantly improves the model-based development process, due to e.g. seamless tool-chains, high automation potentials, consistency between the model and autocode. However, errors are still revealed during an autocode review, despite use of a CG. Therefore, autocode reviews are a valuable and necessary safeguarding technique for automatically generated code cf. [20].

However, for reasons of effort and effectiveness, autocode reviews must be performed differently than reviews of handcrafted C code. The overall review procedure proposed earlier by the authors in [1] and [8] (consisting of (1) combined model and autocode reviews and (2) a two-phase autocode review procedure) has been proven successful in practice. Nevertheless, the focused model review approach needs further improvement. Improved static model analyzers are required in order to optimize manual model review activities.

The reported numbers for review durations allow the review efforts to be estimated in future projects. Furthermore, in combination with recently published effort estimations for other development and QA activities (cf. e.g. [17]), they make it possible to improve the overall model-based development and quality assurance approach.

Complexity metrics for hand code and experiences from its application cannot be simply adapted to autocode. The relationship between model and autocode complexity as well as the influence of CG optimizations on code complexity requires further research activities. Finally, new or at least modified complexity measures for models and autocode need to be developed.

Altogether, the reported specifics of reviews within a model-based development process can be seen as further evidence for the proposition that model-based software development is a development paradigm of its own that needs its own software engineering research area.

6. ACKNOWLEDGEMENTS
The work described was partially performed as part of the IMMOS project funded by the German Federal Ministry of Education and Research (project ref. 01ISC31D). http://www.immos-project.de.

7. REFERENCES
1. Autocode Metrics

<table>
<thead>
<tr>
<th></th>
<th>M1</th>
<th>M2</th>
<th>M3</th>
<th>M4</th>
<th>M5</th>
<th>M6</th>
<th>M7</th>
<th>M8</th>
<th>total</th>
</tr>
</thead>
<tbody>
<tr>
<td># functions</td>
<td>3</td>
<td>2</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>8</td>
<td>10</td>
<td>8</td>
<td>1</td>
</tr>
<tr>
<td># top level inputs / outputs</td>
<td>17 / 8</td>
<td>4 / 5</td>
<td>11 / 9</td>
<td>43 / 9</td>
<td>12 / 4</td>
<td>10 / 10</td>
<td>19 / 8</td>
<td>2 / 1</td>
<td></td>
</tr>
<tr>
<td>cyclomatic complexity CYCMOD</td>
<td>72</td>
<td>57</td>
<td>175</td>
<td>193</td>
<td>45</td>
<td>47</td>
<td>342</td>
<td>94</td>
<td></td>
</tr>
<tr>
<td>subsystem depth</td>
<td>4</td>
<td>3</td>
<td>3</td>
<td>5</td>
<td>2</td>
<td>7</td>
<td>11</td>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>

2. Review Duration

<table>
<thead>
<tr>
<th></th>
<th>M1</th>
<th>M2</th>
<th>M3</th>
<th>M4</th>
<th>M5</th>
<th>M6</th>
<th>M7</th>
<th>M8</th>
<th>total</th>
</tr>
</thead>
<tbody>
<tr>
<td>total time for review (min)</td>
<td>186</td>
<td>48</td>
<td>112</td>
<td>67</td>
<td>48</td>
<td>74</td>
<td>472</td>
<td>35</td>
<td>157</td>
</tr>
<tr>
<td>mean time for review (min)</td>
<td>93</td>
<td>24</td>
<td>56</td>
<td>34</td>
<td>24</td>
<td>25</td>
<td>157</td>
<td>12</td>
<td>-</td>
</tr>
</tbody>
</table>

3. Findings and Changes

<table>
<thead>
<tr>
<th></th>
<th>M1</th>
<th>M2</th>
<th>M3</th>
<th>M4</th>
<th>M5</th>
<th>M6</th>
<th>M7</th>
<th>M8</th>
<th>total</th>
</tr>
</thead>
<tbody>
<tr>
<td># unspecified</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>55</td>
</tr>
<tr>
<td># questions</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>0</td>
<td>11</td>
<td>0</td>
<td>29</td>
</tr>
<tr>
<td># editorial remarks</td>
<td>5</td>
<td>0</td>
<td>2</td>
<td>13</td>
<td>2</td>
<td>5</td>
<td>22</td>
<td>1</td>
<td>56</td>
</tr>
<tr>
<td># uncritical remarks</td>
<td>5</td>
<td>2</td>
<td>5</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>5</td>
<td>1</td>
<td>24</td>
</tr>
<tr>
<td># important remarks</td>
<td>1</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>28</td>
<td>3</td>
<td>58</td>
</tr>
<tr>
<td># critical remarks</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>7</td>
<td>0</td>
<td>14</td>
<td></td>
</tr>
<tr>
<td># findings (total)</td>
<td>12</td>
<td>5</td>
<td>9</td>
<td>15</td>
<td>8</td>
<td>10</td>
<td>74</td>
<td>6</td>
<td>236</td>
</tr>
</tbody>
</table>

Table 1: Metrics of Implementation Model

Table 2: Autocode Metrics