# An Agile Quality Assurance Framework for the Development of Fusion Real-time Applications

André C. Neto, Filippo Sartori, Riccardo Vitelli, Llorenç Capellà, Giuseppe Ferrò, Ivan Herrero and Héctor Novella

***Abstract*—In the context of a fast controller prototype project, which aimed at testing the integration of fast plant systems in the ITER software environment, a software Quality Assurance (QA) strategy that is appropriate for the development of ITER real-time applications (e.g. diagnostic control systems) is being developed. In particular the QA processes had to be designed in order to safely integrate contributions from a large and heterogeneous development community, which includes developer profiles both from the scientific community and from the industrial suppliers. Notably the coding standard aims at demonstrating MISRA-C++:2008 compliance.**

**The MARTe software framework is currently being used to implement a large variety of fusion real-time control system applications. Being a modular and multi-platform framework has allowed to reuse components, interfaces and services across systems which are deployed in very distinct architectures. This has leveraged the exposure of the same code to different environment configurations, thus increasing the confidence on its quality and robustness.**

**The QA processes are being applied to the development of a new version of the MARTe framework. The main objective is to provide a QA certifiable environment from where it is possible to develop, with less effort, certifiable applications. This is expected to be achieved by sharing the same QA methodologies and tools and by maximising the reuse of framework modules (which were also developed against these QA processes).**

**This paper details the QA processes, the associated tools and discusses its applicability to the fusion development environment.**

***Index Terms*—Software quality assurance, real-time, control software, MARTe.**

## I. Introduction

SOFTWARE frameworks are usually associated with the need to satisfy common and transversal requirements in a given development context. When a project chooses to take advantage of an existing software framework, the development team expectation is to be able to save effort in the full development cycle domain, i.e. from design to integration. As a consequence, a software framework is expected to be robust and to behave coherently with its specifications. This requires the existence of appropriate and complete documentation, complemented by thorough testing which exercises the framework in all of the anticipated use-cases. Moreover, it is also crucial to have a mechanism which guarantees that any defects found in the framework can be properly signalled and managed by the framework developers. All of the above can only be appropriately handled by guaranteeing that the framework has a sound quality assurance process associated to it.

The MARTe software framework [1] is C++ modular and multi-platform framework for the development of real-time control system applications. As of today, it has been deployed in many fusion real-time control systems [2], particularly in the JET tokamak [3]. One of the main advantages of the MARTe architecture is the bold separation between the platform specific implementation, the environment details and the real-time algorithms (i.e. the user code). The platform is defined by the target processor and the operating system, while the environment encapsulates all the interfacing details which are related to the peculiarities of the location where the final system is to be deployed. This includes both the interfacing with the hardware plaform and the binding to the services that allow to configure and retrieve data from the system. This clear separation of concern has allowed to reuse many components inside the same environment (e.g. all the systems deployed at JET share the same services for parameter configuration and data retrieval) and to develop and test the user algorithms in non-real-time operating systems and to later deploy the same exact code in a previously tested platform.

As more systems started to use MARTe, the number of supported environments and platforms has considerably grown. This has leveraged the exposure of the same core code to different environment configurations, thus increasing the confidence on its quality and robustness. Having the same infrastructure being used inside a community has also had the advantage of sharing and recycling knowledge about the framework and its architecture. In the context of an internal ITER fast controller [4] prototype project, which aimed at testing the integration of fast plant systems in the ITER software environment, a new version of the MARTe framework has been developed. One of the main objectives of this activity was to develop a software Quality Assurance (QA) strategy that is appropriate for the development of ITER real-time applications (e.g. diagnostic control systems). In particular the QA process had to be designed in order to safely integrate contributions from a large and heterogeneous development community, which includes developer profiles both from the scientific community and from the industrial suppliers.

Manuscript received May 29, 2016. The work leading to this publication has been funded partially by Fusion for Energy under the Contract F4E-OFC-361-06. This publication reflects the views only of the author, and Fusion for Energy cannot be held responsible for any use which may be made of the information contained therein.

A. C. Neto, F. Sartori and R. Vitelli are with Fusion for Energy, Josep Pla 2 08019 Barcelona, Spain (e-mail: Andre.Neto@f4e.europa.eu).

L . Capellà is with Vitrociset Belgium, Rue Devant les Hetres 2, Belgium

G . Ferrò is with Dipartimento di Ingegneria Civile e Ingeneria Informatica. Universit di Roma Tor Vergata, Via del Politecnico 1. 00133 Rome, Italy

I. Herrero and H. Novella are with GTD Sistemas de Información, 08005 Barcelona, Spain.

## II. Project organisation

As discussed before, one of the main objectives of this exercise was to enable a development which would enable external contributions from a very heterogeneous community, without compromising the overall quality of the project. As it can be seen in Fig. 1 the selected model comprises three main actors. The brainstorming group discusses and proposes requirements for MARTe. The coordinator decides which requirements are feasible and relevant for the framework and integrates them into the developing documentation. Finally, the developing team designs and implements the software accordingly to these requirements. It should be noted that it is possible to share members between the brainstorming community in the development team, provided they strictly abide to the QA processes.

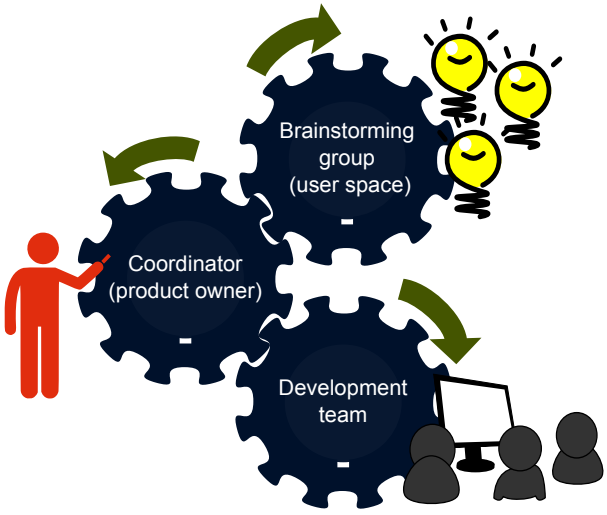


Fig. 1. Top level project organization. The brainstorming group proposes requirements which are filtered by the coordinator and later delegated to the development team.

### A. Deliverables

The main project deliverables are listed in table I. In particular, with each release of the framework, a new version of each of these deliverables is to be issued. The requirements deliverable describes the features that the framework must meet and its constraints. The architecture and design deliverable details the system architecture and its design. It contains a model of the main blocks of the framework, the expected interactions between components and the links between the requirements and the components. The software deliverable comprises the source code and API documentation, as well the unit tests and integration tests source code. The QA audit deliverable contains all the quality reviews made for the other deliverables and all the detected non-conformities. Finally, the traceability matrix shows the links between the requirements and the design classes.

### B. Software activities

As shown in Fig. 2, the software activities are structured following a double-V-model approach where the first V (blue) is related to the development activities of the framework source-code and the second (red) implements the MARTe quality assurance. The double-V emphasises the fact that each of the activities has its traceable mirror.

| Deliverable | Name scheme | Type |
|---|---|---|
| **Requirements** | MARTe requirements (vX.Y) | Report |
| **Architecture & design** | MARTe architecture & design (vX.Y) | Report |
| **Software** | | |
| Source code | MARTe source code (vX.Y) | Source |
| API documentation | MARTe API documentation (vX.Y) | Report |
| Unit tests | MARTe unit tests (vX.Y) | Source |
| Integration tests | MARTe integration tests (vX.Y) | Source |
| **QA audit** | MARTe QA-audit (vX.Y) | Report |
| **Traceability matrix** | MARTe traceability matrix (vX.Y) | Report |

TABLE I
MARTe deliverables for each release, where X denotes a major version and Y a minor version.

The requirements activity translates the needs and objectives for MARTe, as defined by the stakeholders, into a proper set of requirements. The output is a model of requirements expressed in UML, which are grouped into fields of interest using UML packages. The review process verifies that all requirements have been correctly written, following a given set of rules, and defined in the requirements document.

The architecture & design activity is driven from the requirements defined in the previous activity. The design includes a subset of classes considered the architectural foundation of the framework, complemented by any other classes which are needed for the framework to work. The review process verifies that the architecture and design are aligned with requirements and follow best practices. The responsible has to pay special attention to possible redundant modules and into the global coherence of system.

The code and documentation review process includes the manual verification of the API documentation and the verification that a set of best practices were followed during the code implementation. This activity is complemented by the manual verification of the API documentation generated from the comments in the source-code and by the execution of an automated tool which verifies the compliance of the source-code against the coding standard.

The software activities are concluded with the development of unit and integration tests. The unit test classes will implicitly trace the implementation classes of the source code, because it is assumed that each unit test class tests only one implementation class. The integration tests aim at exercising the architecture classes in the widest set of representative use-cases. The review of the unit tests is divided in a static and in a dynamic analysis phase. In the former, the reviewer verifies how many public functions of the source code have unit tests defined (black box unit testing is assumed). The latter, calculates what percentage of code has been executed (white/grey box unit testing is assumed). In both cases, code with a low coverage percentage ($< 80\%$) will be rejected.

## III. Quality assurance

As depicted in Fig. 3, the QA system is described in five different documents (which combined provide an integral

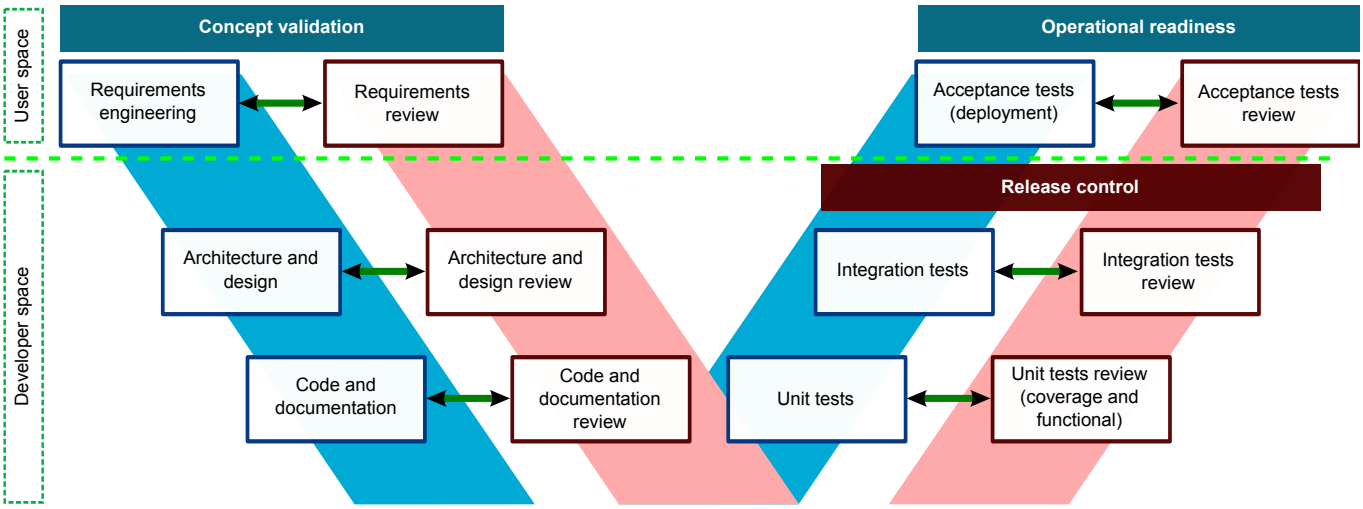


Fig. 2. The blue V is in charge of developing the MARTe framework. The red V guarantees the framework quality assurance. It should be noted that this double-V-model does not prescribe any given software lifecycle methodology.

processes). The Project Management Plan (PMP) details the software development process described above.


Fig. 3. Relationship between the QA plans. Quality, configuration management and verification are integral processes.

The Quality Assurance Plan (QAP), establishes the process and procedures that are used to achieve the objectives of the quality assurance process. In particular, it defines the role of the Quality Officer (QO) and its main responsibilities, which include independent reviews and audits of all the data and processes involved in the development, production, and maintenance of the deliverables. The QO also verifies the degree of compliance against the project applicable standards, project plans and processes. This plan also imposes the template format of the audit documentation (inputs, outputs and checklists) and sets up a procedure for process improvement.

The Configuration Management Plan (CMP) identifies the items which will be under configuration management and establishes the strategy for baseline and change control. The CMP defines that *Git* [5] is used as the version control system for the software development and imposes the workflow (described in section V).

The Verification and Validation Plan (VVP) details the verification processes applied by the project to satisfy the software verification process objectives, including software testing, reviews and traceability. Specifically, it imposes that the verification team shall not have taken part in the development process, in order to ensure the independence between development and verification activities. Nevertheless, the VVP allows for the exchanging of roles during the life-cycle of the project (i.e. a developer may be a reviewer of another development activity, on which this developer was not involved). The VVP also defines the transition criteria between software activities (see Fig. 4) and the auditing templates in a format compatible with the issue tracking tool (detailed in section V).

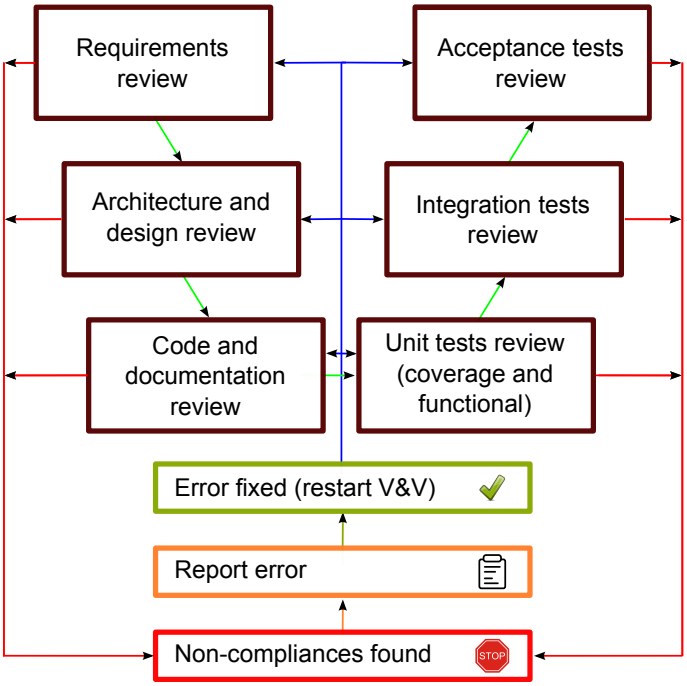


Fig. 4. All the possible states and transitions of the verification and validation phases.

The Coding Standard document defines the main rules applicable to the source code development. The main goal of these

rules is to assure the coherence among the code developed by the different team members and to also assure its quality and maintainability. In order to increase the robustness of the code and to avoid common errors and pitfalls, a controlled subset of the C++ language was defined for the MARTe framework. This subset is defined by means of a list of coding rules, which will address many of the dangerous aspects of the C++ language for critical systems. Thus, the C++ version used on MARTe will be that defined by the standard ISO/IEC 14882:2003 [6] (also known as C++03), while the coding rules will be those defined by the standard MISRA C++:2008 [7]. The MISRA C++:2008 is a set of software development guidelines for the C++ language targeted towards critical systems, which applies to the C++ language defined by the standard ISO/IEC 14882:2003. MISRA C++:2008 has emerged from the automotive industry, and is widely accepted as a model for best practices in sectors like aerospace, telecom, medical devices, defense, railway and others. MISRA C++:2008 also takes into account all those features of the language that are unspecified and undefined in the ISO/IEC 14882:2003.

All the coding rules must be followed by developers at coding time, but formal reviews are also performed in order to ensure that all written code conforms to them. This is mostly performed by means of a static code checking tool (described later) which automatically detects violations to the rules in the code. For those rules which are not automatable, a manual review is performed before any major release. The output of this process is traced in a compliance matrix which will list all the rules and duly justifies any deviations. The coding standard also defines and imposes rules and guidelines for documentation, naming and code formatting.

## IV. Software lifecycle implementation

In order to implement the software activities described in section II-B, the software lifecycle is managed using a mixture between a waterfall and an *Agile* [8] approach. Each user-story aims at developing the components required to satisfy at least one of the framework requirements. These user-stories live in the product backlog and at the beginning of every sprint, a subset is selected to be implemented in the scope of the sprint. During the sprint planning meeting the user-stories for the sprint are set in order of priority. During the sprint implementation, the development team moves the user-stories into the quality process depicted in Fig. 2. Each of the steps shown in Fig. 4 has to be reviewed by the quality responsible before the user-story is allowed to move forward in the V-model. This means that while a user-story is being reviewed (e.g. code review) the developer is allowed to start (or continue) working in another story (e.g. the test implementation of a story which had its code reviewed already). Once a user-story successfully passes the integration test review, all of its components are set to be released with the next version of the framework.

Following the spirit of an *Agile* development, the development team aims at meeting and reporting daily on the development status. Each sprint is concluded with a sprint review where the team demonstrates a potentially shippable product increment (i.e. that at least one of the user-stories has successfully passed the integration test review).

The process optimisation discussed before is implemented during the sprint review meeting, where the team members are asked to openly identify issues, which are then classified into one of the following three types: (i) things that the team should start doing; (ii) things that the team should stop doing; and (iii) things that the team should continue doing. Approved process improvements are then implemented by updating the appropriate standards, procedures, processes, checklists, or other related documentation (which are controlled configuration items). Software defects and other non-compliances are also translated into user-stories which follow this same QA process.

## V. Development tools

The processes described above are supported by a set of tools which aim at assisting the developer at consistently meeting the coding standard rules, the quality reviewer at maximising the number of automatic verifications and the project manager at having a sound overview of the project status.

Source cove versioning is performed using *Git* and *GitLab*[9]. The *Git* workflow, which based in the *Git* branching model presented in [10], is described in Fig. 5 where it shows that each user-story is always created from the development branch. This branch is only merged back into development if it successfully passes all the quality checks. At the end of the sprint the development branch, which includes all the finalised user-stories, is merged into the release branch. All the quality auditing is performed over this branch and minor bug fixing is allowed. Finally, the release branch is merged into the master branch and a new tag with the version number created. It should be noted that after being merged into develop branch, the user-story feature is deleted. An *hotfix* branch is used to resolve critical bugs in the master version. When resolved, the branch will be merged back into the master and develop branches (if applicable it will also be merged to the release branch).

The *Redmine* [11] issue tracking system is used to manage the *Agile* workflow and to store all the quality reports, including the audits and the *Agile* sprint planning and review meeting minutes. More specifically, each user-story is assigned to a redmine issue and its life-cycle managed using the Redmine Agile plugin [12] (see example in Fig. 6). The plug-in was configured in order to provide a one-to-one mapping with the V-model, greatly simplifying the management and overview of the QA review process.

Unit testing is performed using the *google-test* framework [13]. Nevertheless, and given that MARTe is also expected to be deployed in bare-metal systems (i.e. processors without an operating system), the *google-test* framework is only used as a front-end engine to execute the MARTe unit test class methods, which are written without any dependencies on the *google-test* framework. This allows for the same tests to be easily ported to another testing framework. Code coverage is implemented with *gcov*, the *GNU Project Compiler Collection* [14] coverage testing tool and is complemented with a graphical front-end named *lcov* [15].

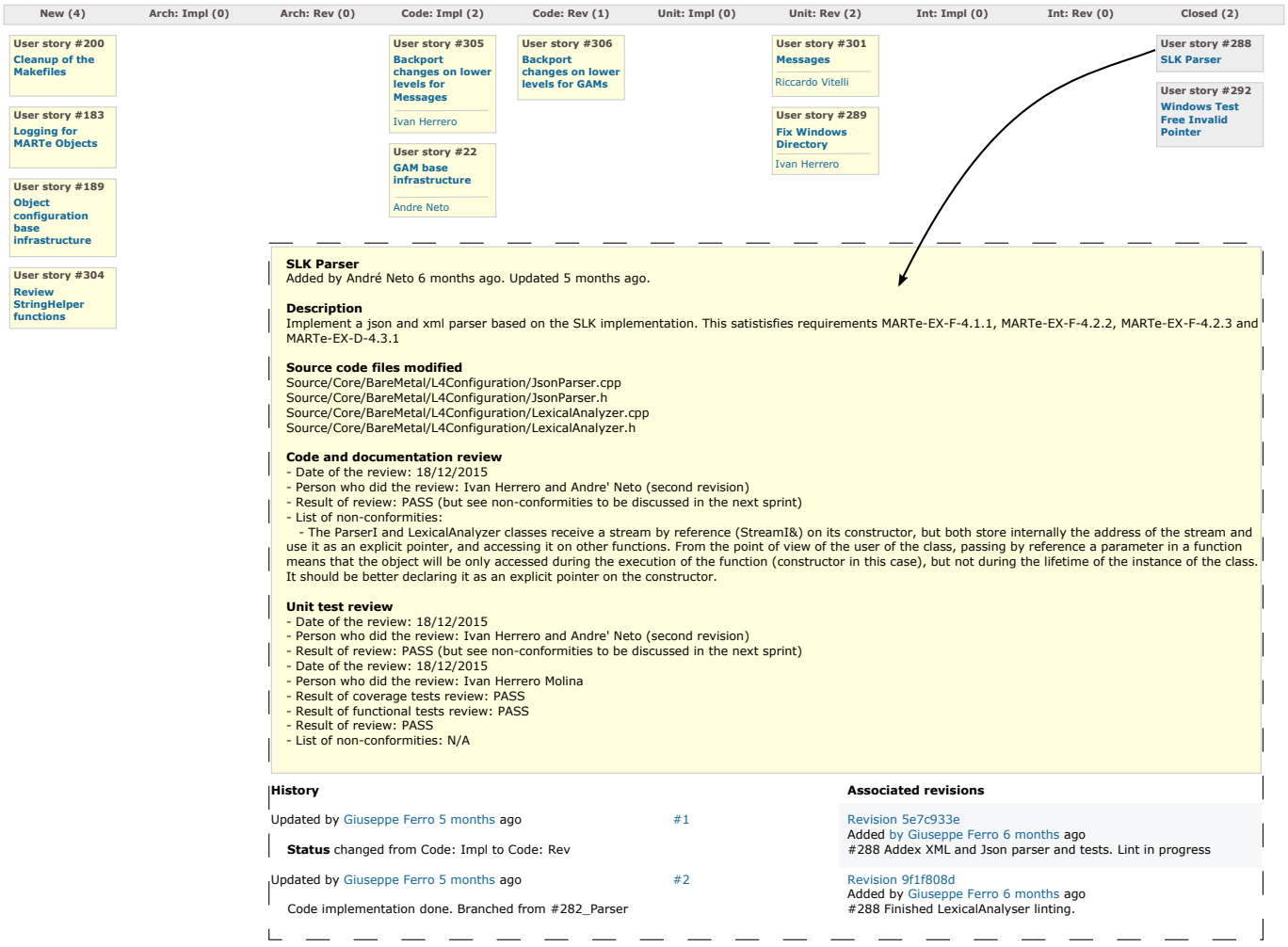


Fig. 6. Snapshot of the *Agile* board. Each user-story is associated to a *Redmine* issue where its QA lifecycle is fully audited.

Static code analysis and compliance to the MISRA C++:2008 standard is performed using the *Gimpel Software FlexeLint* tool [16]. Common project deviations to the standard are added to a file that is shared among all the developers. Specific deviations to the standard are directly justified in the source-code using a special syntax inside a C++ comment. Code is expected to be developed using the *Eclipse CDT* [17] integrated development environment (IDE). *Eclipse* is configured with the project formatting, code and editor templates, as specified in the verification and validation plan. The unit tests can also be executed from with-in the IDE by using the *Google Tests Runner*. The source-code documentation is generated using *Doxygen* [18] and is integrated into the IDE through the *eclox* plug-in [19]. The output of the execution of *FlexeLint* has been also integrated into the editor, allowing developers to quickly navigate to the non-compliant lines of code and to use the static analysis tool as part of their development process. Finally, the complet management of the *Git* workflow can also be performed directly from with-in the IDE.

### A. Software integration environment

The continuous integration environment is implemented using the *Jenkins* software [20]. This is configured to build and test the development version (i.e. from the *Git* develop branch) of the framework every night. In particular it reports on the testing coverage (using the *cobertura* [21], *sonar* [22] and *JUnit* reporting plug-ins) and on the percentage of compliance generated by the static code analysis tool.

## VI. Lessons learned

As of today there were already 11 development releases of the framework following the QA system. There are approximately 30000 lines of framework code and around 45000 lines of testing code.

The compliance to the coding standard (and to the selected MIRSA rules) is of almost 100 %. Adjusting *Flexelint* requires time and expertise. A misconfigured static analysis tool will generate a large number of false positives and might fail to identify true errors. Given that the standard is very demanding the amount of *linting* errors on any given file can be very large (this number decreases significantly with experience). Not having the output of the tool directly integrated into the IDE would greatly increase the development time and possible jeopardise the usage of the tool. It should be noted that most of these *linting* errors would not prevent the compiler from building without generating any warnings. Moreover, being

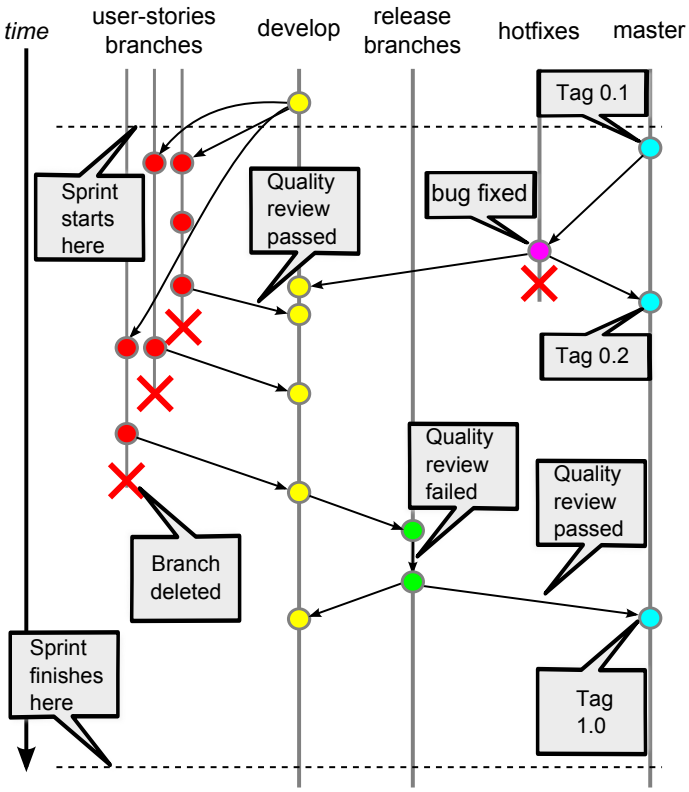


Fig. 5. The chosen *Git* workflow, which is based on the branching model presented in [10].

able to justify the deviations to the standard in the source-code increases the transparency of the QA process and facilitates the review process.

Regarding the unit tests, more than 2500 tests are currently being executed with a coverage in excess of 90 %. The code sections which are most difficult to test are the ones related to operating system or processor faults. Not using all the functionalities of the *google-test* framework limits the amount of features that could be used to facilitate the identification of errors. On the other side, being able to execute all the tests of the framework in a bare-metal system (based on the ARM ® Cortex ®-M [23] has proved to be crucial for this type of development. Driven by the limitation in the amount of development effort available, it was decided not to apply the coding standard to the development of the test files. Documentation is of extreme importance and is the process with less room for automation. As a consequence a large part of the reviewing time is spent on verifying the documentation.

Concerning the QA processes, associating each user-story to a new git branch has proven to be crucial in order to be able to allocate and monitor the work in a controlled way. Having the user-story following the V-model allows to have full transparency on who is doing what. Even if the model has been working very successfully, the fact that the user-story has to be reviewed several times during its life-cycle could possibly be optimised and further aligned to the *Agile* model. In such case, the reviewers would only verify the finalised user-story after the developers had finished the coding and testing phases. Indeed, it already happened that a user-story, upon test implementation, has required to change the original code, consequently triggering the review process to start again.

Having the user-stories reviewed using the same structure of the QA audit greatly simplifies the writing of the final auditing. The *Redmine* agile board has also allowed to greatly simplify the management of the user-stories and its associated life cycle. Aligning sprint with releases has also proven to work very well and allows to have all the QA reports consistent and with the same release numbers of the software.

## VII. Conclusions

The QA framework presented in this paper can be easily adapted to the development of many types of software which are common in the fusion community, in particular for software related to control and data acquisition systems that is to be shared among different teams.

Even if the effort required to setup the QA system is not negligible, the major challenge is to make sure that it is applied throughout the full development process. This necessarily requires some extra-effort, but using some of the automation techniques described above the review time can be significantly optimised. In particular the deployment of a static analysis tool, fully integrated in the development editor, greatly increases the confidence that a large set of the coding rules was correctly applied. The obvious price of not following a complete QA system, is that it will be very difficult to distribute and maintain the software, at least without the continuous support of the original developers, which in turn also has a significant trade-off cost associated to it.

## References

[1] A. C. Neto, F. Sartori, F. Piccolo, R. Vitelli, G. De Tommasi, L. Zabeo, A. Barbalace, H. Fernandes, D. F. Valcarcel, and A. J. N. Batista, "Marte: A multiplatform real-time framework," *Nuclear Science, IEEE Transactions on*, vol. 57, no. 2, pp. 479 –486, april 2010.

[2] A. C. Neto, D. Alves, L. Boncagni, P. J. Carvalho, D. F. Valcarcel, A. Barbalace, G. De Tommasi, H. Fernandes, F. Sartori, E. Vitale, R. Vitelli, and L. Zabeo, "A survey of recent marte based systems," *IEEE Transactions on Nuclear Science*, vol. 58, no. 4, pp. 1482–1489, 2011.

[3] D. Alves, A. C. Neto, D. F. Valcrcel, R. Felton, J. M. Lpez, A. Barbalace, L. Boncagni, P. Card, G. D. Tommasi, A. Goodyear, S. Jachmich, P. J. Lomas, F. Maviglia, P. McCullen, A. Murari, M. Rainford, C. Reux, F. Rimini, F. Sartori, A. V. Stephen, J. Vega, R. Vitelli, L. Zabeo, and K. D. Zastrow, "A new generation of real-time systems in the jet tokamak," in *Real Time Conference (RT), 2012 18th IEEE-NPSS*, June 2012, pp. 1–9.

[4] A. Wallander, L. Abadie, H. Dave, F. Di Maio, H. Gulati, C. Hansalia *et al.*, "ITER instrumentation and controlstatus and plans," *Fusion Engineering and Design*, vol. 85, no. 3 - 4, pp. 529 – 534, 2010.

[5] git, https://git-scm.com/, 5/26/2016.

[6] *Programming Languages—C++*, 2003, ISO/IEC 14882:2003(E).

[7] *MISRA C++:2008 Guidelines for the Use of the C++ Language in Critical Systems*, 2008, ISBN 978-906400-03-3 (paperback), ISBN 978-906400-04-0 (PDF).

[8] The Scrum Guide, https://www.scrumalliance.org/why-scrum/scrum-guide, 5/25/2016.

[9] GitLab, https://about.gitlab.com/, 5/26/2016.

[10] A successful Git branching model, http://nvie.com/posts/a-successful-git-branching-model/, 5/26/2016.

[11] Redmine, http://www.redmine.org/, 5/26/2016.

[12] Scrum and Agile project management plugin for redmine, http://www.redminecrm.com/, 5/25/2016.

[13] Google Test, https://github.com/google/googletest, 5/26/2016.

[14] gcov, https://gcc.gnu.org/onlinedocs/gcc/Gcov.html, 5/26/2016.

[15] LCOV - the LTP GCOV extension, http://ltp.sourceforge.net/coverage/lcov.php, 5/26/2016.
[16] FlexeLint for C/C++, http://www.gimpel.com/html/flex.htm, 5/25/2016.
[17] Eclipse CDT (C/C++ Development Tooling), https://eclipse.org/cdt/, 5/26/2016.
[18] Doxygen, http://www.stack.nl/ dimitri/doxygen/, 5/26/2016.
[19] Eclox, http://home.gna.org/eclox/, 5/26/2016.
[20] Jenkins, https://jenkins.io/, 5/26/2016.
[21] Cobertura, http://cobertura.github.io/cobertura/, 5/26/2016.
[22] SonarQube, http://http://www.sonarqube.org/, 5/26/2016.
[23] G. Ferrò, A. C. Neto, F. Sartori, L. Boncagni, D. Carnevale, M. Gospodarczyk, A. Monti, A. Moretti, R. Vitelli, L. Capella, and I. Herrero, "Embedded implementation of a real-time switching controller on a robotic arm," in *Proc. 20th IEEE-NPSS Real Time Conf. (RT)*, 2016.