Software Requirements Specification for an Engineering Computational Tool

2024-06-14

Abstract

An imaginary (a thought experiment if you will) "Request for Quotation" issued by a fictitious agency asking for vendors to offer a free and open source cloud-based computational tool to solve engineering problems. This (imaginary but plausible) Software Requirements Specification document describes the mandatory features this tool ought to have and lists some features which would be nice the tool had, following current state-of-the-art methods and technologies.

Contents

1 Introduction **2**
- 1.1 Objective . . . 2
- 1.2 Scope . . . 3

2 Architecture **3**
- 2.1 Deployment . . . 4
- 2.2 Execution . . . 4
- 2.3 Efficiency . . . 4
- 2.4 Scalability . . . 5
- 2.5 Flexibility . . . 5
- 2.6 Extensibility . . . 5
- 2.7 Interoperability . . . 5

3 Interfaces **5**
- 3.1 Problem input . . . 5
- 3.2 Results output . . . 6

4 Quality assurance **6**
- 4.1 Reproducibility and traceability . . . 6
- 4.2 Automated testing . . . 7
- 4.3 Bug reporting and tracking . . . 7
- 4.4 Verification . . . 8
- 4.5 Validation . . . 9
- 4.6 Documentation . . . 9

1 Introduction

A computational tool (herein after referred to as *the tool*) specifically designed to be executed in arbitrarily-scalable remote servers (i.e. in the cloud) is required in order to solve engineering problems following the current state-of-the-art methods and technologies impacting the high-performance computing world. This (imaginary but plausible) Software Requirements Specification document describes the mandatory features this tool ought to have and lists some features which would be nice the tool had. Also it contains requirements and guidelines about architecture, execution and interfaces in order to fulfill the needs of cognizant engineers as of the 2020s.

On the one hand, the tool should allow to solve industrial problems under stringent efficiency (sec. 2.3) and quality (sec. 4) requirements. It is therefore mandatory to be able to assess the source code for

- independent verification, and/or
- performance profiling, and/or
- quality control

by qualified third parties from all around the world. Hence, it has to be *open source* according to the definition of the Open Source Initiative.

On the other hand, the initial version of the tool is expected to provide a basic functionality which might be extended (sec. 1.1 and sec. 2.6) by academic researchers and/or professional programmers. It thus should also be *free*—in the sense of freedom, not in the sense of price—as defined by the Free Software Foundation. There is no requirement on the pricing scheme, which is up to the vendor to define in the offer along with the detailed licensing terms. These should allow users to solve their problems the way they need and, eventually, to modify and improve the tool to suit their needs. If they cannot program themselves, they should have the *freedom* to hire somebody to do it for them.

1.1 Objective

The main objective of the tool is to be able to solve engineering problems which are usually casted as differential-algebraic equations (DAEs) or partial differential equations (PDEs), such as

- heat conduction
- mechanical elasticity
- structural modal analysis
- mechanical frequency studies
- electromagnetism
- chemical diffusion
- process control dynamics
- computational fluid dynamics
- …

on one or more mainstream cloud servers, i.e. computers with hardware and operating systems (further discussed in sec. 2) that allows them to be available online and accessed remotely either interactively or automatically by other computers as well. Other architectures such as high-end desktop personal computers or even low-end laptops might be supported but they should not the main target (i.e. the tool has to be cloud-first but laptop-friendly).

The initial version of the tool must be able to handle a subset of the above list of problem types. Afterward, the set of supported problem types, models, equations and features of the tool should grow to include other

models as well, as required in sec. 2.6.

1.2 Scope

The tool should allow users to define the problem to be solved programmatically. That is to say, the problem should be completely defined using one or more files either…

a. specifically formatted for the tool to read such as JSON or a particular input format (historically called input decks in punched-card days), and/or
b. written in an high-level interpreted language such as Python or Julia.

Once the problem has been defined and passed on to the solver, no further human intervention should be required.

It should be noted that a graphical user interface is *not* required. The tool may include one, but it should be able to run without needing any interactive user intervention rather than the preparation of a set of input files. Nevertheless, the tool might *allow* a GUI to be used. For example, for a basic usage involving simple cases, a user interface engine should be able to create these problem-definition files in order to give access to less advanced users to the tool using a desktop, mobile and/or web-based interface in order to run the actual tool without needing to manually prepare the actual input files.

However, for general usage, users should be able to completely define the problem (or set of problems, i.e. a parametric study) they want to solve in one or more input files and to obtain one or more output files containing the desired results, either a set of scalar outputs (such as maximum stresses or mean temperatures), and/or a detailed time and/or spatial distribution. If needed, a discretization of the domain may to be taken as a known input, i.e. the tool is not required to create the mesh as long as a suitable mesher can be employed using a similar workflow as the one specified in this SRS.

The tool should define and document (sec. 4.6) the way the input files for a solving particular problem are to be prepared (sec. 3.1) and how the results are to be written (sec. 3.2). Any GUI, pre-processor, post-processor or other related graphical tool used to provide a graphical interface for the user should integrate in the workflow described in the preceding paragraph: a pre-processor should create the input files needed for the tool and a post-processor should read the output files created by the tool.

2 Architecture

The tool must be aimed at being executed unattended on remote servers which are expected to have a mainstream (as of the 2020s) architecture regarding operating system (GNU/Linux variants and other Unix-like OSes) and hardware stack, such as

- a few Intel-compatible or ARM-like CPUs per host
- a few levels of memory caches
- a few gigabytes of random-access memory
- several gigabytes of solid-state storage

It should successfully run on

- bare-metal
- virtual servers
- containerized images

using standard compilers, dependencies and libraries already available in the repositories of most current operating systems distributions.

Preference should be given to open source compilers, dependencies and libraries. Small problems might be executed in a single host but large problems ought to be split through several server instances depending on the processing and memory requirements. The computational implementation should adhere to open and well-established parallelization standards.

Ability to run on local desktop personal computers and/laptops is not required but suggested as a mean of giving the opportunity to users to test and debug small coarse computational models before launching the large computation on a HPC cluster or on a set of scalable cloud instances. Support for non-GNU/Linux operating systems is not required but also suggested.

Mobile platforms such as tablets and phones are not suitable to run engineering simulations due to their lack of proper electronic cooling mechanisms. They are suggested to be used to control one (or more) instances of the tool running on the cloud, and even to pre and post process results through mobile and/or web interfaces.

2.1 Deployment

The tool should be easily deployed to production servers. Both

a. an automated method for compiling the sources from scratch aiming at obtaining optimized binaries for a particular host architecture should be provided using a well-established procedures, and
b. one (or more) generic binary version aiming at common server architectures should be provided.

Either option should be available to be downloaded from suitable online sources, either by real people and/or automated deployment scripts.

2.2 Execution

It is mandatory to be able to execute the tool remotely, either with a direct action from the user or from a high-level workflow which could be triggered by a human or by an automated script. Since it is required for the tool to be able to be run distributed among different servers, proper means to perform this kind of remote executions should be provided. The calling party should be able to monitor the status during run time and get the returned error level after finishing the execution.

The tool shall provide means to perform parametric computations by varying one or more problem parameters in a certain prescribed way such that it can be used as an inner solver for an outer-loop optimization tool. In this regard, it is desirable that the tool could compute scalar values such that the figure of merit being optimized (maximum temperature, total weight, total heat flux, minimum natural frequency, maximum displacement, maximum von Mises stress, etc.) is already available without needing further post-processing.

2.3 Efficiency

As required in the previous section, it is mandatory to be able to execute the tool on one or more remote servers. The computational resources needed from this server, i.e. costs measured in

- CPU/GPU time
- random-access memory
- long-term storage

- etc.

needed to solve a problem should be comparable to other similar state-of-the-art cloud-based script-friendly finite-element tools.

2.4 Scalability

The tool ought to be able to start solving small problems first to check the inputs and outputs behave as expected and then allow increasing the problem size up in order to achieve to the desired accuracy of the results. As mentioned in sec. 2, large problem should be split among different computers to be able to solve them using a finite amount of per-host computational power (RAM and CPU).

2.5 Flexibility

The tool should be able to handle engineering problems involving different materials with potential spatial and time-dependent properties, such as temperature-dependent thermal expansion coefficients and/or non-constant densities. Boundary conditions must be allowed to depend on both space and time as well, like non-uniform pressure loads and/or transient heat fluxes.

2.6 Extensibility

It should be possible to add other problem types casted as PDEs (such as the Schröedinger equation) to the tool using a reasonable amount of time by one or more skilled programmers. The tool should also allow new models (such as non-linear stress-strain constitutive relationships) to be added as well.

2.7 Interoperability

A mean of exchanging data with other computational tools complying to requirements similar to the ones outlined in this document. This includes pre and post-processors but also other computational programs so that coupled calculations can be eventually performed by efficiently exchanging information between calculation codes.

3 Interfaces

The tool should be able to allow remote execution without any user intervention after the tool is launched. To achieve this goal it is required that the problem should be completely defined in one or more input files and the output should be complete and useful after the tool finishes its execution, as already required. The tool should be able to report the status of the execution (i.e. progress, errors, etc.) and to make this information available to the user or process that launched the execution, possibly from a remote location.

3.1 Problem input

The problem should be completely defined by one or more input files. These input files might be

a. particularly formatted files to be read by the tool in an *ad-hoc* way, and/or
b. source files for interpreted languages which can call the tool through and API or equivalent method, and/or
c. any other method that can fulfill the requirements described so far.

Preferably, these input files should be plain ASCII files in order to allow to manage changes using distributed version control systems such as Git. If the tool provides an API for an interpreted language such as Python, then the Python source used to solve a particular problem should be Git-friendly. It is recommended not to track revisions of mesh data files but of the source input files, i.e. to track the mesher's input and not the mesher's output. Therefore, it is recommended not to mix the problem definition with the problem mesh data.

It is not mandatory to include a GUI in the main distribution, but the input/output scheme should be such that graphical pre and post-processing tools can create the input files and read the output files so as to allow third parties to develop interfaces. It is recommended to design the workflow as to make it possible for the interfaces to be accessible from mobile devices and web browsers.

It is expected that 80% of the problems need 20% of the functionality. It is acceptable if only basic usage can be achieved through the usage of graphical interfaces to ease basic usage at first. Complex problems involving non-trivial material properties and boundary conditions not be treated by a GUI and only available by needing access to the input files.

3.2 Results output

The output ought to contain useful results and should not be cluttered up with non-mandatory information such as ASCII art, notices, explanations or copyright notices. Since the time of cognizant engineers is far more expensive than CPU time, output should be easily interpreted by either a human or, even better, by other programs or interfaces—especially those based in mobile and/or web platforms. Open-source formats and standards should be preferred over privative and ad-hoc formatting to encourage the possibility of using different workflows and/or interfaces.

4 Quality assurance

Since the results obtained with the tool might be used in verifying existing equipment or in designing new mechanical parts in sensitive industries, a certain level of software quality assurance is needed. Not only are best-practices for developing generic software such as

- employment of a version control system,
- automated testing suites,
- user-reported bug tracking support.
- etc.

required, but also since the tool falls in the category of engineering computational software, verification and validation procedures are also mandatory, as discussed below. Design should be such that governance of engineering data including problem definition, results and documentation can be efficiently performed using state-of-the-art methodologies, such as distributed control version systems

4.1 Reproducibility and traceability

The full source code and the documentation of the tool ought to be maintained under a control version system. Whether access to the repository is public or not is up to the vendor, as long as the copying conditions are compatible with the definitions of both free and open source software from the FSF and the OSI, respectively as required in sec. 1.

In order to be able to track results obtained with different version of the tools, there should be a clear release procedure. There should be periodical releases of stable versions that are required

- not to raise any warnings when compiled using modern versions of common compilers (e.g. GNU, Clang, Intel, etc.)
- not to raise any errors when assessed with dynamic memory analysis tools (e.g. Valgrind) for a wide variety of test cases
- to pass all the automated test suites as specified in sec. 4.2

These stable releases should follow a common versioning scheme, and either the tarballs with the sources and/or the version control system commits should be digitally signed by a cognizant responsible. Other unstable versions with partial and/or limited features might be released either in the form of tarballs or made available in a code repository. The requirement is that unstable tarballs and main (a.k.a. trunk) branches on the repositories have to be compilable. Any feature that does not work as expected or that does not even compile has to be committed into develop branches before being merge into trunk.

If the tool has an executable binary, it should be able to report which version of the code the executable corresponds to. If there is a library callable through an API, there should be a call which returns the version of the code the library corresponds to.

It is recommended not to mix mesh data like nodes and element definition with problem data like material properties and boundary conditions so as to ease governance and tracking of computational models and the results associated with them. All the information needed to solve a particular problem (i.e. meshes, boundary conditions, spatially-distributed material properties, etc.) should be generated from a very simple set of files which ought to be susceptible of being tracked with current state-of-the-art version control systems. In order to comply with this suggestion, ASCII formats should be favored when possible.

4.2 Automated testing

A mean to automatically test the code works as expected is mandatory. A set of problems with known solutions should be solved with the tool after each modification of the code to make sure these changes still give the right answers for the right questions and no regressions are introduced. Unit software testing practices like continuous integration and test coverage are recommended but not mandatory.

The tests contained in the test suite should be

- varied,
- diverse, and
- independent

Due to efficiency issues, there can be different sets of tests (e.g. unit and integration tests, quick and thorough tests, etc.) Development versions stored in non-main branches can have temporarily-failing tests, but stable versions have to pass all the test suites.

4.3 Bug reporting and tracking

A system to allow developers and users to report bugs and errors and to suggest improvements should be provided. If applicable, bug reports should be tracked, addressed and documented. User-provided suggestions might go into the back log or TO-DO list if appropriate.

Here, “bug and errors” mean failure to

- compile on supported architectures,
- run (unexpected run-time errors, segmentation faults, etc.)
- return a correct result

4.4 Verification

Verification, defined as

> The process of determining that a model implementation accurately represents the developer's conceptual description of the model and the solution to the model.

i.e. checking if the tool is solving right the equations, should be performed before applying the code to solve any industrial problem. Depending on the nature and regulation of the industry, the verification guidelines and requirements may vary. Since it is expected that code verification tasks could be performed by arbitrary individuals or organizations not necessarily affiliated with the tool vendor, the source code should be available to independent third parties. In this regard, changes in the source code should be controllable, traceable and well documented.

Even though the verification requirements may vary among problem types, industries and particular applications, a common method to verify the code is to compare solutions obtained with the tool with known exact solutions or benchmarks. It is thus mandatory to be able to compare the results with analytical solutions, either internally in the tool or through external libraries or tools. This approach is called the Method of Exact Solutions and it is the most widespread scheme for verifying computational software, although it does not provide a comprehensive method to verify the whole spectrum of features. In any case, the tool's output should be susceptible of being post-processed and analyzed in such a way to be able to determine the order of convergence of the numerical solution as compared to the exact one.

Another possibility is to follow the Method of Manufactured Solutions, which does address all the shortcomings of MES. It is highly encouraged that the tool allows the application of MMS for software verification. Indeed, this method needs a full explanation of the equations solved by the tool, up to the point that a report from Sandia National Labs says that

> Difficulties in determination of the governing equations arises frequently with commercial software, where some information is regarded as proprietary. If the governing equations cannot be determined, we would question the validity of using the code.

To enforce the availability of the governing equations, the tool has to be open source as required in sec. 1 and well documented as required in sec. 4.6.

A report following either the MES and/or MMS procedures has to be prepared for each type of equation that the tool can solve. The report should show how the numerical results converge to the exact or manufactured results with respect to the mesh size or number of degrees of freedom. This rate should then be compared to the theoretical expected order.

Whenever a verification task is performed and documented, at least one of the cases should be added to the test suite. Even though the verification report must contain a parametric mesh study, a single-mesh case is enough to be added to the test suite. The objective of the tests defined in sec. 4.2 is to be able to detect regressions which might have been inadvertently introduced in the code and not to do any actual verification. Therefore a single-mesh case is enough for the test suites.

4.5 Validation

As with verification, for each industrial application of the tool there should be a documented procedure to perform a set of validation tests, defined as

> The process of determining the degree to which a model is an accurate representation of the real world from the perspective of the intended uses of the model.

i.e. checking that the right equations are being solved by the tool. This procedure should be based on existing industry standards regarding verification and validation such as ASME, AIAA, IAEA, etc. There should be a procedure for each type of physical problem (thermal, mechanical, thermomechanical, nuclear, etc.) and for each problem type when a new

- geometry,
- mesh type,
- material model,
- boundary condition,
- data interpolation scheme

or any other particular application-dependent feature is needed.

A report following the validation procedure defined above should be prepared and signed by a responsible engineer in a case-by-case basis for each particular field of application of the tool. Verification can be performed against

- known analytical results, and/or
- other already-validated tools following the same standards, and/or
- experimental results.

4.6 Documentation

Documentation should be complete and cover both the user and the developer point of view. It should include a user manual adequate for both reference and tutorial purposes. Other forms of simplified documentation such as quick reference cards or video tutorials are not mandatory but highly recommended. Since the tool should be extendable (sec. 2.6), there should be a separate development manual covering the programming design and implementation, explaining how to extend the code and how to add new features. Also, as non-trivial mathematics which should be verified are expected, a thorough explanation of what equations are taken into account and how they are solved is required.

It should be possible to make the full documentation available online in a way that it can be both printed in hard copy and accessed easily from a mobile device. Users modifying the tool to suit their own needs should be able to modify the associated documentation as well, so a clear notice about the licensing terms of the documentation itself (which might be different from the licensing terms of the source code itself) is mandatory. Tracking changes in the documentation should be similar to tracking changes in the code base. Each individual document ought to explicitly state to which version of the tool applies. Plain ASCII formats should be preferred. It is forbidden to submit documentation in a non-free format.

The documentation shall also include procedures for

- reporting errors and bugs
- releasing stable versions
- performing verification and validation studies
- contributing to the code base, including

- code of conduct
- coding styles
- variable and function naming conventions