SITA Version 0

A simulation and code testing assistant for TOUGH2 and MARNIE

GRS - 400

Gesellschaft für Anlagen-
und Reaktorsicherheit
(GRS) gGmbH

SITA
Version 0

A simulation and code testing assistant for TOUGH2 and MARNIE

Holger Seher
Martin Navarro

June 2016

Remark:

This study has been funded by the German Federal Ministry for the Environment, Nature Conservation, Building and Nuclear Safety (BMUB) under the support codes UM13A03400 and 3614R03200.

The work was conducted by Gesellschaft für Anlagen- und Reaktorsicherheit (GRS) mbH.

The authors are responsible for the content of the report.

GRS - 400
ISBN 978-3-944161-81-5

Key Words:
SITA, quality assurance, TOUGH2, MARNIE

Abstract

High quality standards have to be met by those numerical codes that are applied in long-term safety assessments for deep geological repositories for radioactive waste. The software environment SITA (“*a **si**mulation and code **t**esting **a**ssistant for TOUGH2 and MARNIE”*) has been developed by GRS in order to perform automated regression testing for the flow and transport simulators TOUGH2 and MARNIE. GRS uses the codes TOUGH2 and MARNIE in order to assess the performance of deep geological repositories for radioactive waste. With SITA, simulation results of TOUGH2 and MARNIE can be compared to analytical solutions and simulations results of other code versions. SITA uses data interfaces to operate with codes whose input and output depends on the code version. The present report is part of a wider GRS programme to assure and improve the quality of TOUGH2 and MARNIE. It addresses users as well as administrators of SITA.

Contents

1 Introduction

Numerical simulations are an essential part of assessing the long-term safety of deep geological repositories for radioactive waste. The codes that are used as simulators have to meet high quality standards in order to support the trustworthiness of the safety assessment. Therefore, it is necessary that the quality of the code is demonstrated and the measures of quality assurance are well documented.

One aspect of quality assurance is the validation of the code, i. e. the check whether the numerical implementation of physical models reflects the intended physics. There cannot be any absolute validation of a code because it cannot be proved that every possible application of the code will reflect the intended physics. Yet, code tests can be performed in large number and will help to demonstrate and improve the quality of the code, especially if the test cases have been selected carefully. Validation tests should not only cover standard but also very special and rare system states which might activate parts or states of the code which are seldom encountered and thus less tested.

Verification tests do not only require a large number of test cases but also a frequent repetition of tests. The code has to be verified before the release of every new code version. There may also be a desire to perform tests even more frequently, perhaps after every milestone of code development. The need for frequent code tests as well as the large number of test cases calls for an automation of the testing procedure.

For the analysis of flow and transport processes in deep geological repository systems, GRS uses the code MARNIE, which is an in-house development of GRS, as well as codes based on the TOUGH2 code, which has been developed by the Lawrence Berkeley National Laboratories /PRU 99/.

MARNIE /MAR 02/ simulates the transport of water, brine and constituents on grids that are constructed by the linkage of 1-dimensional sub-grids in the 3-dimensional space. Although applicable to repositories in different types of host rock, MARNIE focusses on processes that are relevant to repositories in salt formations.

The TOUGH2 code simulates multiphase flow and transport of liquid water, vapour, non-condensable gases, and heat in porous and fractured media. GRS works with different versions and modifications of the TOUGH2 code like, for example, the code TOUGH2-

GRS /NAV 13/ which incorporates additional processes relevant to repository systems and which is still further developed. GRS also employs the parallel version TOUGH2-MP /ZHA 08/.

In order to perform automatic testing for MARNIE and TOUGH2 codes, GRS has developed the software environment SITA. SITA is “*a **si**mulation and code **t**esting **a**ssistant for TOUGH2 and MARNIE”* written in Perl /PERL 14/. SITA is only an “assistant” because code testing cannot be automated in whole, and still the interpretation of simulation results affords expert judgment. Besides being a testing assistant, SITA also serves as a simulation environment for TOUGH2 and MARNIE. The present report presents the SITA version 0.1.a (see chapter 3.2 for an explanation of version numbers).

SITA allows the user to compare simulation results to analytical solutions and results of other code versions. This poses that problem of running version-independent verification tests with codes that have a version-dependent input and output. SITA solves this problem by introducing data interfaces for each code version.

SITA is part of a larger bundle of quality assurance measures for the codes TOUGH2-GRS and MARNIE, which are described in report /HOT 16/. The development of SITA is linked to the GRS software development project for quality assurance of codes /GRS 13/ and has been carried out in two projects:

- **Project ZIESEL**

  UM13A03400, Forschung und Entwicklung zum Zweiphasenfluss in einem salinaren Endlager am Beispiel des ERAM

  In this project, the SITA concept, the SITA implementation for TOUGH2 based codes, and this documentation were developed by Martin Navarro and Holger Seher. SITA was restructured to an object oriented design by Holger Seher using the package Moose for Perl.

- **Project EMIL**

  3614 R 03200, Forschung und Entwicklung zu Methoden und Instrumenten des Langzeitsicherheitsnachweises

  In this project, SITA was upgraded for the use with MARNIE by Holger Seher. (Postprocessing for TOUGHREACT is not fully available yet).

The present report serves as a manual for code testers, test case developers, code developers and SITA administrators:

- **SITA administrators**
  should be familiar with the administration of UNIX/Linux systems.
  Recommended chapters:
  - chapter 2 (overview)
  - chapter 4 (installation)
  - chapter 7 (creation of make files)

- **Code testers who do not create own test cases**
  should be familiar with the code they want to test (TOUGH2 or MARNIE).
  Recommended chapters:
  - chapter 2 (overview)
  - chapter 3 (source code management)
  - chapter 0 (sita options and features)

- **Code developers who change the input or output format of the code or the compilation procedure**
  should be familiar with *gnu make* /GNU 14/.
  Recommended chapters:
  - chapter 2 (overview)
  - chapter 3 (source code management)
  - chapter 7 (creation of make files)
  - chapter 9 (input and output interfaces)

- **Test case developers**
  should be familiar with the JSON data format /JSON 13/ (see appendix A)
  Recommended chapters:
  - chapter 2 (overview)
  - chapter 3 (source code management)
  - chapter 8 (test cases)
  - chapter 9 (input interfaces)

2 Overview

SITA is a software environment mainly designed for automatic regression tests. Although constructed for the codes TOUGH2 and MARNIE, the concept of SITA is flexible and allows the integration of other codes.

SITA is designed for code projects which use the software versioning and revision control system SVN (Apache Subversion), whose application is part of the code quality assurance of GRS. The SITA development project itself uses SVN too.

SITA requires that the source codes belonging to a code are organized in a folder tree. The name of the top directory refers to the code (e.g. TOUGH2 or MARNIE). This folder can contain several variants of the source codes, which may represent different stages of code development or different levels of quality assurance. These source code variants are stored in sub folders named trunk, branches and tags (see chapter 3 for details). We will call all folders which hold one complete set of source code files a source code folder. SITA gives simple access to all source code folders. SITA offers aliases for long path names.

Fig. 2.1 illustrates how SITA works together with the local working copy of the SVN repository. Each source code folder includes a *makefile* for the compilation of executables, a short description of the code version, and two data interfaces that give SITA the ability to handle version-dependent input and output. SITA interacts with auxiliary programmes like Fortran compilers, *GNU make* and *GNU plot* and generates output in **H**yper**t**ext **M**arkup **L**anguage (HTML) format.

SITA runs code tests but also serves as a pre- and post-processing tool. Code tests are defined by test case files (see chapter 8). They contain the simulation input as well as user defined analyses in JSON format. Analyses definitions instruct SITA to retrieve simulation results, which can then be compared to expected values and displayed in graphical or textual form. Analyses definitions can also instruct SITA to compare simulation results of different executables. For this reason, a single test case file can trigger more than one simulation.

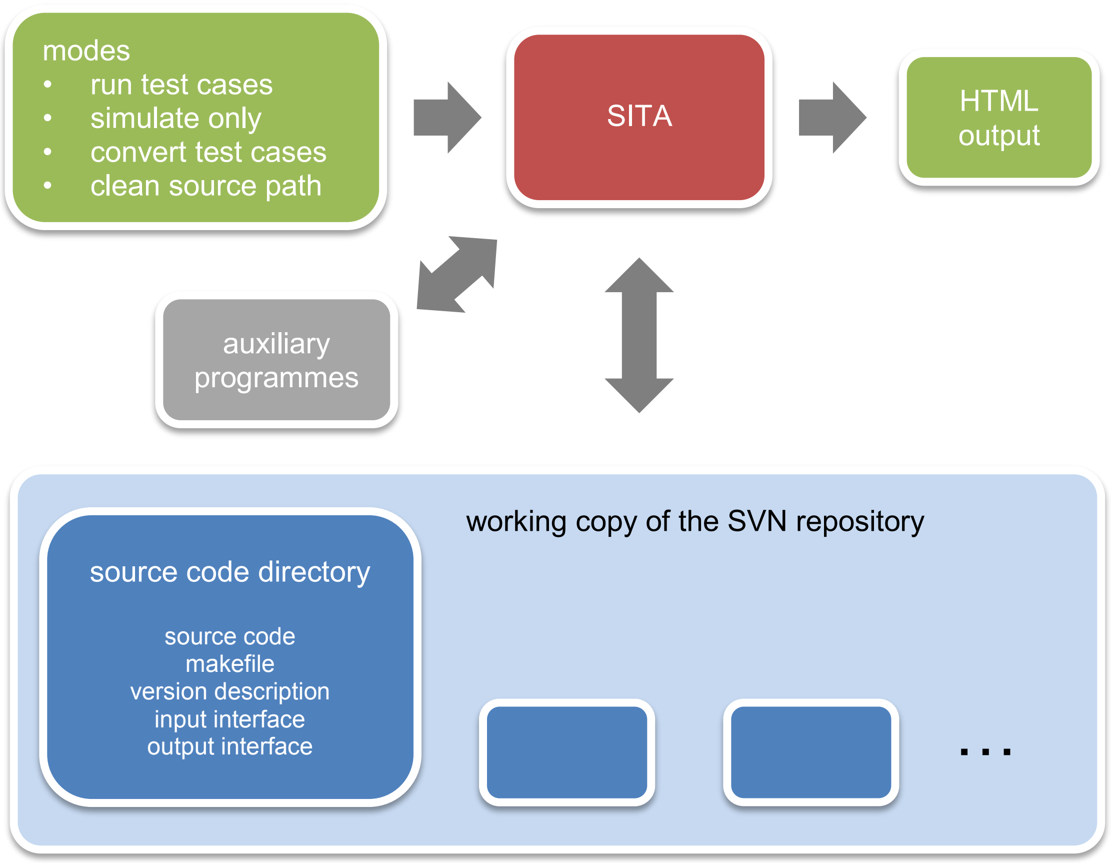

Fig. 2.1 Environment of SITA

3 Source code management

At GRS, all variants of a source code are stored in a repository on an SVN server. Users have local working copies of this repository on their computers. SITA can easily access every specific code family, code, code version, source code, and executable inside the folder tree of the local repository copy.

Variants of a source code are stored in hierarchical order (Fig. 3.1). The top folder refers to the *code family* (e.g. TOUGH2-based codes) and there are separate subfolders for each *code* (e.g. TOUGH2-GRS or MARNIE). Each code folder is subdivided in folders called *trunk*, *branches*, *tags*, and *tags_not_QA*, which refer to a specific developer or degree of quality assurance. This aspect will be explained in detail in the following chapters. The actual *source code* is located in the low-level folders. Every source code can generate one or more *executables*. The terms *code*, *code version*, *source code* and *executable* will be defined in more detail in the following chapters.

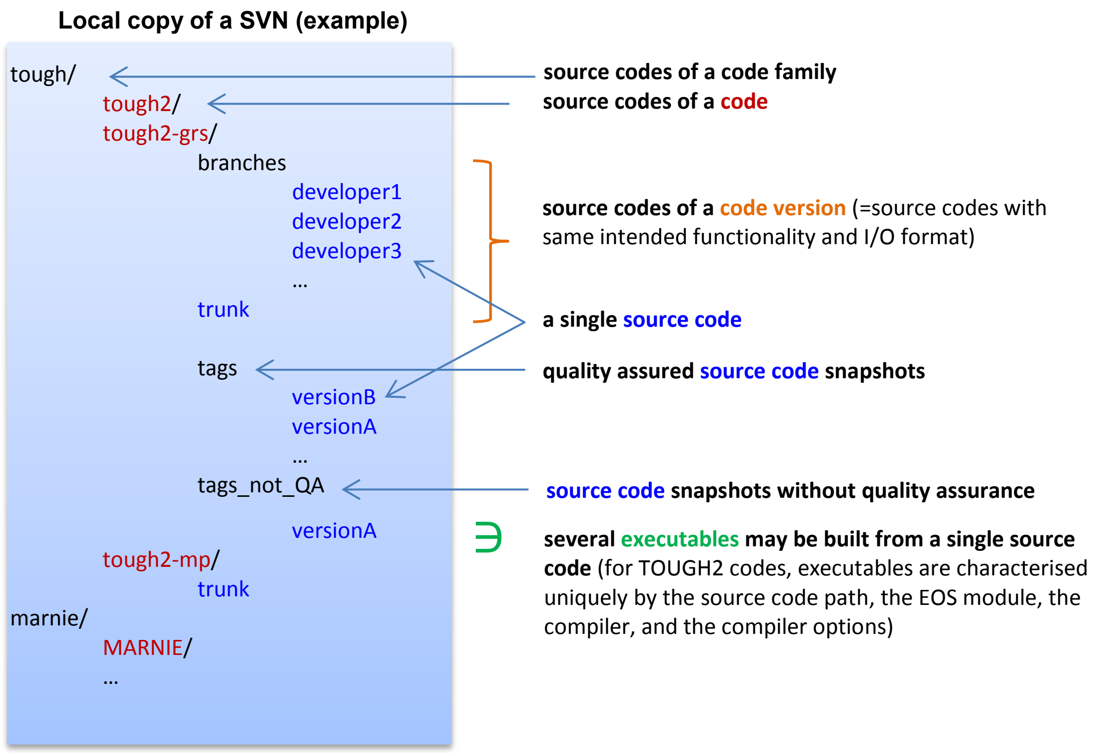
Local copy of a SVN (example)
tough/
tough2/
tough2-grs/
branches
developer1
developer2
developer3
…
trunk
tags
versionB
versionA
…
tags_not_QA
versionA
tough2-mp/
trunk
marnie/
MARNIE/
…
source codes of a code family
source codes of a code
source codes of a code version (=source codes with same intended functionality and I/O format)
a single source code
quality assured source code snapshots
source code snapshots without quality assurance
∋
several executables may be built from a single source code (for TOUGH2 codes, executables are characterised uniquely by the source code path, the EOS module, the compiler, and the compiler options)

Fig. 3.1 Example structure of a folder tree holding variants of TOUGH2 or MARNIE source codes

3.1 Codes

Within the framework of SITA, a *code* is defined as an independent set of *source codes* (see section 3.3) including different *code versions* (see section 3.2). For example, “TOUGH2”, “TOUGH2-GRS” and “TOUGH2-MP” are perceived as separate codes and are therefore stored in separate code directories. In the following example of a local code repository, the code directories are marked in boldface:

```
tough/
   tough2/
          …
   tough2-grs/
          branches
                 developer1
                 developer2
                 …
          trunk
          tags
          tags_not_QA
   tough2-mp/
```

```
                trunk
        …
```

Code directories split up into subdirectories (`trunk`, `tags`, `branches`, `tags_not_QA`) which help to distinguish between different code developers, code versions, or levels of quality assurance.

3.2 Major versions, minor versions, and patch versions

The TOUGH2-GRS and MARNIE projects distinguish between major code versions, minor code versions and patch versions. Version names of TOUGH2-GRS comply with the format X.Y.Z where X refers to the major version (numerical value), Y to a minor version (numerical value) and Z to the patch version (lower-case letter). The first version is 0.1.a. Version names of MARNIE use the same format except for the missing second dot (e.g. "10.3a").

The shift to a new patch, minor, or major version is governed by the following definitions and rules:

- The ***major code version*** is related to specific code development projects. The major version number is increased at the start of the project and the code documentation is updated when the project finishes. There are no requirements on how much the code function has to change in order to justify a new major code version.
- The ***minor code version*** relates to code compatibility. Source codes that share the same minor version have to be compatible, which requires that
  - input and output parameters are the same and have the same meaning
  - the intended code function is the same

  This ensures that failing code comparisons (failing simulations or significant deviations of simulation results) indicate errors or inaccuracies of the compared codes instead of intended function changes. There is also no guarantee that codes with differing minor version can be compared.

  Minor code versions have been introduced because the functionality of a code changes gradually while the next major code version is developed. Consequently, test cases for one source code might not be applicable to another. Test cases have to determine to which source codes they apply. This is done by means of the minor version number and the documented function changes between minor code versions. A test case can be applied to all source codes sharing the same minor version number. It may also be applicable to other minor code versions, but this cannot be guaranteed.

  Note that pure changes of the input or output format do not justify a change to a new minor code version. SITA is able to deal with changes of this kind if the input and output interfaces (see chapter 9) remain up-to-date.

- **Patch versions** introduce bugfixes or minor code changes. Successive patch versions have to be compatible. If this is not the case, the minor code version number has to be increased (again starting with patch "a").

3.3 Source codes

A source code is a complete set of source code files. Each source code is stored in a separate folder. The position of the source code folders in the repository's folder tree is shown in the following example in boldface:

```
tough/
   tough2-grs/
          branches
                 developer1
                 developer2
                 …
          trunk
          tags
                 tag1
          tags_not_QA
                 tag1_not_QA
```

In the QA system of GRS, the `trunk` folder holds the main not quality assured development line of the upcoming code version. Each developer has an own source code folder in the `branches` folder. Usually, branches are copies of the trunk, which are reintegrated on a regular basis. Tags are quality assured snapshots of the trunk. In contrast, snapshots without quality assurance are stored in `tags_not_QA`.

Each source code directory has to be supplemented by four files:

- Every source code has a major, minor, and path version. This information is stored in the file `version.json`, which is located in the source code directory (see chapter 9.1).

- A *makefile* to build executables (see chapter 7).
- At least two interface files describing the specific format of the input and output (it is possible to introduce EOS specific interfaces for TOUGH2). SITA needs these files to generate code input and to read the code's output (see chapter 9).

3.4 Executables

A single source code may generate more than one executable. For example, a TOUGH2 source code can be compiled with different EOS modules. On the other hand, MARNIE can be built with different numbers of so called p-modules. Also different compiler choices or compiler options will lead to different executables.

SITA has to identify executables in a unique way. This is done using the following information:

- the *directory path* containing the source code (and the code name and version if the directory holds several source code subdirectories)
- the *name of the EOS module* (for TOUGH2) or the *number of p-modules* (for MARNIE)
- the *compiler* to use, and
- the applied *compiler options*.

The directory path is specified relatively to the repository's base path. If the specified directory does not contain any version file (and thus no source code), SITA follows the subdirectories in order to find a source code fitting to the specified code name and code version. SITA offers aliases for long path names. These can be found in the file `aliases.json`, which is located in the repository's base directory. Calling SITA with the help option will list all available aliases.

3.5 What is tested?

Code tests are designed to evaluate the quality of a code or of a code version but, strictly speaking, code tests are only tests of particular executables. A TOUGH2 source code can generate different executables depending on choice of the compiler, compiler options, external libraries and platform. Usually, not all of these executables are subject to quality assurance. It is therefore important to document how a quality assured executable has been generated. Therefore, SITA includes this information in the generated test results.

4 Installation

The installation steps described in chapter 4.1 have to be performed by the system administrator. Installations which only require user rights are described in chapter 4.2.

4.1 Required programmes

SITA is written in Perl 5 and has been designed for Cygwin/Linux/Unix platforms, which are used as development platforms at GRS. SITA uses auxiliary programmes and packages, which have to be installed first (see Tab. 4.1; please refer to the respective manuals for proper installation and configuration).

Tab. 4.1 Auxiliary programmes and shell commands used by SITA

Programmes	command	Function
`Perl 5, version 22 or higher` with `JSON` and `Moose` packages		Perl interpreter for the execution of SITA
`GNU Make`	`make`	compiles executables from the source code using “makefiles”
`GNU Fortran` or `Intel Fortran`	`gfortran` or `ifort`	compiles Fortran code
`gnuplot 5.0`	`gnuplot`	generates postscript plots
Ghostscript	`gs`	converts postscript plots to bitmaps
`pdfTex` with `amsmath` and `siunitx` packages	`latex`	for displaying formulas in the HTML output of SITA
`ImageMagick`	`convert`	used to convert the pdf output of `Latex` into an image that can be displayed in the HTML output of SITA
`torque, moab`	`qstat`, `qsub`, `showstart`	job control and scheduling used on the GRS HPC linux cluster (optional)
`GNU Fortran`	`gcov`	determines code coverage for gfortran (optional)
`Intel Fortran`	`profmerge`, `codecov`	determines code coverage for ifort (optional)
	`copy`, `mkdir`, `rm`, `cd`	shell commands
	`sed`, `grep`	text processing
	`|`, `>`, `tee`	redirects output channels

SITA uses the job control and scheduling commands of the programs `torque` and `moab`, which are used by the queuing system of the GRS high-performance computing (HPC) linux cluster. The presence of the scheduling system is checked by SITA by checking the availability of the `qsub` command.

Unless it has already been installed, the local Perl installation has to be extended by the **J**ava**S**cript **O**bject **N**otation (JSON) package, which can be downloaded from the **C**omprehensive **P**erl **A**rchive **N**etwork (CPAN, http://www.cpan.org). It is sufficient to store the package files in the `…\lib\perl5\X.YZ` directory.

SITA uses Moose package which is an extension of the Perl 5 object system. However, it is not sufficient to just copy the Moose package into a directory. Moose has to be installed using the command

```
perl -MCPAN -e"install Moose"
```

In order to install Moose on Cygwin, the following Cygwin packages have to be installed:

- `CPAN-Meta-YAML`
- `gcc-core`
- `gcc-fortran`
- `gcc-g++`
- `libcrypt-devel (cygwin 64 bit)`
- `crypt (Cygwin 32 bit)`
- `perl-JSON`
- `perl-YAML`
- `perl-CPAN-Meta-YAML`
- `perl-ExtUtils-CBuilder`
- `perl-ExtUtils-Depends`
- `perl-ExtUtils-F77`
- `perl-ExtUtils-LibBuilder`
- `perl-ExtUtils-MakeMaker`
- `perl-ExtUtils-PkgConfig`
- `make`

4.2 Environment variables

In order to run SITA, the following environment variables have to be set (if the *bash* shell is used, the environment variables can be set in the `~/.bashrc` file).

- The root path of SITA (holding the programmes `t2sita.pl` and `masita.pl`) has to be added to the `PATH` and the `PERL5LIB` variable. Since SITA is stored in an own SVN repository, the root path of SITA points to the root path of the local repository copy. Example:

  ```
  export PATH=$PATH:~/repo/sita
  ```

  ```
  export PERL5LIB=$PERL5LIB:~/repo/sita
  ```

- `T2GRS` has to be set to the root path of the local repository copy holding the TOUGH2 or MARNIE project. Example: `export T2GRS=~/repo/tough`

- `T2TESTS` has to be set to the user's standard directory for TOUGH2 test cases. Example: `export T2TESTS =~/repo/toughtests/trunk`

- `MARNIEPATH` has to be set to the root path of the local repository copy holding the MARNIE project. Example: `export MARNIEPATH=~/repo/marnie`

- `MARNIETESTS` holds the user's standard directory for MARNIE test cases. Example: `export MARNIETESTS=~/repo/marnietests/trunk`

Now, the programmes `t2sita.pl` and `masita.pl` for TOUGH2 and MARNIE, respectively, can be called from the console.

5 Structure of the TOUGH2 input and output

Prior to the description of test cases (chapter 8) and input and output interfaces (chapter 9), we give a brief overview over the structure of the input and output of the TOUGH2 based codes.

5.1 Input

The input interface describes the syntax of a TOUGH2 input file. A TOUGH2 input file is made of data blocks (or "cards"), like `ROCKS` (for material properties) and `MULTI` (specifies the number of fluid components and balance equations) blocks. Each block starts with a keyword (e.g. `"ROCKS"`, `"MULTI"`). The following lines contain input data in fixed format. Most data blocks have to close with a blank line. Example:

```
ROCKS----1----*----2----*----3----*----4----*----5----*----6----*----7----*----8
UHALF    2 2.400E+03    1.E-01    1.E-15    1.E-15    1.E-15   2.5E-01 1.000E+03
        0.
    6             0.
    8
LHALF    2 2.400E+03    1.E-01    1.E-15    1.E-15    1.E-15   2.5E-01 1.000E+03
        0.
    6             0.
    8

MULTI----1----*----2----*----3----*----4----*----5----*----6----*----7----*----8
    3    3    2    6    0
```

There are optional and mandatory blocks and data entries. While optional data blocks may be omitted, optional data entries have to be substituted by blank strings of the same length in order to keep the following columns at the proper place.

Some data blocks may contain recurring lines and line groups. For example, the material parameters of the `ROCK` block has to be repeated for each material. This means that the input interface has to allow for repetitions (see also section 9.2.4).

5.2 Output

TOUGH2 uses several output channels. State and flow printouts as well as time step information are usually written to the standard output channel STDOUT (except for TOUGH2-MP, which creates the files `OUTPUT` and `OUTPUT_DATA`). If SITA calls TOUGH2, it directs the TOUGH2 output to the STDOUT channel to the main output file `OUTFILE`.

TOUGH2-GRS also creates the files `ELE_MAIN` and `CON_MAIN` for the element and connection specific data, respectively, that is printed to STDOUT. These two files use the data format of the time series files `FOFT`, `COFT`, and `GOFT` (see below).

Time series are printed to the files `FOFT`, `COFT`, and `GOFT` (evolution of states, flows and generation rates, respectively). TOUGH2-GRS also generates a file `DOFT` for domain specific quantities (TOUGH2 domains are also called *materials* or *rocks*). If the input parameter `KDATA` of data block `PARAM` equals 4, an extended set of output parameters is used for the `COFT` and `DOFT` files.

Output to STDOUT

Every time step, TOUGH2 produces a short printout containing the current time, time step, number of iterations, and so forth. This printout has the form

```
A11 1(     1,  2) ST = 0.100000E+02 DT = 0.100000E+02 …
```

where `ST` is the simulation time and `DT` the time step width.

The main printouts of the system state cover states and fluxes for all elements and connections, respectively. These printouts are more complex. They start with a header like

```
          OUTPUT DATA AFTER (    0,   0)-1-TIME STEPS …

 @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ …

  TOTAL TIME      KCYC    ITER  ITERC                …
 0.000000E+00       0       0      0                 …

 @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ …
```

which is followed by several output blocks. The first output block

```
ELEM. INDEX            P          Sliq …
                    [Pa]           [1] …
 A11 1     1 0.10000E+07 0.80000E+00 …
 A21 1     2 0.10000E+07 0.80000E+00 …
…
```

is reserved for element specific output parameters. The second (optional) output block

```
     ELEM1  ELEM2  INDEX     FLOH      …
                              (W)      …
     A11 1  A21 1      1 0.000000E+00 …
     A21 1  A31 1      2 0.000000E+00 …
```

```
…
```

holds information on advective phase and heat fluxes. The third (again optional) output block

```
ELEM1 ELEM2  PHASE COMP  PHASE COMP  PHASE COMP  …
               -1-  -1-    -1-  -2-    -1-  -3   …
                                                 …
 AAA 1 AAA 2 0.00000E+00 0.00000E+00 0.00000E+00 …
 AAA 1 AAA 3 0.00000E+00 0.00000E+00 0.00000E+00 …
…
```

lists diffusive fluxes of components. The number of output blocks is controlled by the TOUGH2 input parameter `KDATA`. The used EOS module determines the set of output parameters.

After the last block, there is a short printout of component masses in active elements:

```
MASS IN PLACE
 GAS 0.583413E+01 KG;     LIQUID 0.498781E+03 KG;   BRINE 0.000000E+00 …
 AIR 0.590252E+01 KG
```

The form of this printout depends on the EOS module.

Due to the fixed column format used by TOUGH2 printouts, the number of bytes per printout and per output block is always the same. This includes that the file position of location printouts (i.e. a printout line for a specific element or connection) relative to the printout header is always the same. SITA uses this fact to find location printouts. Therefore, the user has to take care that printouts as well as the contained blocks have the same length if new code versions are created.

Time series files

Time series for selected elements or connections are stored in separate files like e.g. `FOFT`, `COFT`, `GOFT`, or `DOFT`. Every line of such a file is a comma-separated list of numbers:

```
time step, time, location index, P1, P2, … , PN, location index, P1, P2, … , PN, …
```

The first line entry is the time step number followed by the simulation time in seconds. A series of sub-lists for selected locations follows. Every sub-list starts with the element or connection index (an integer) followed by a constant set of output parameters *P1* to *PN* (reals).

In the `FOFT`, `COFT`, `GOFT`, and `DOFT` data blocks, the user specifies which locations he wants to print. The distinctive feature of the `DOFT` data block is that the *first* location entry will appear as the *second* location in the `DOFT` file because the first location in the `DOFT` file is reserved for the entire grid model (only active elements).

ELE_MAIN and CON_MAIN (only TOUGH2-GRS)

The `ELE_MAIN` and `CON_MAIN` files produced by TOUGH2-GRS follow the same syntax as the time series files. The only difference is that the time steps and times correspond to those of the main printout to the STDOUT channel. The status of all elements and connections is listed.

6 Using SITA

This chapter gives a brief overview over the commands and concept of SITA. It will give sufficient information for SITA users that want to run existing test case files. Users who want to compile test case files on their own should continue to read chapter 8, which describes the structure of test cases files.

6.1 SITA options

SITA offers the two commands `t2sita.pl` and `masita.pl` for the TOUGH2 and MARNIE source code, respectively. SITA is controlled by command options of the form `-option` and `-option value`. Some options are code specific, and there are some that are mandatory. Tab. 6.1 gives a complete list of SITA options.

If SITA is called with the `-help` option, a help text will be displayed. In all other cases the `-mode` option has to be provided in order to determine SITA's general mode of operation:

- `-mode test` performs code tests by means of test case files. Generally speaking, a test case file combines simulation input with instructions on how to display simulation results. The latter are written to the html file `sita.html` in the current directory. Test case files are written in JSON format. Their structure is described in chapter 8 in detail.

  SITA's test mode requires additional options. The `-in` option specifies the test case or a list of test cases. The `-sc` option (or alternatively the `-path` and `-eos` options for TOUGH2 or the `-path` and `-npmodule` options for MARNIE) specifies the source code.

  Examples:

  ```
  t2sita.pl -mode test -path tough2/trunk -eos eos7 -in mytest.json

  t2sita.pl -mode test –sc "tough2/trunk eos7" -in mytest.json

  masita.pl -mode test –path marnie/trunk -in mytest.json –npmodule 400

  masita.pl -mode test –sc "matrunk 400" -in mytestlist.list
  ```

- `-mode run` starts a TOUGH2 or MARNIE simulation using the code specific input files. The mandatory options of mode `run` are the same as for mode `test`.
- `-mode convert` extracts TOUGH2 or MARNIE input files from SITA test case files. Mode `convert` has the same mandatory options as mode `test`.
- In mode `test`, SITA can analyse the code coverage of simulations. The option `-mode clean` removes the code coverage information and object files which have been generated by SITA in the source code directory during code compilation. The path of the source code directory has to be provided by the option `-path`.

Tab. 6.1 SITA options (default values underlined)

Default options are underlined. Letters 'M' and 'O' on the right hand side of the table stand for 'mandatory' and 'optional', respectively. 'M*' and 'O*': the `-sc` options replaces the `-path` and `-eos/-npmodule` options.

`-help`	Displays the help text				
`-mode` **`test`**	Executes SITA tests cases (HTML output)				
`-mode` **`run`**	Executes TOUGH2 or MARNIE input files (code specific output)				
`-mode` **`convert`**	Converts SITA test cases to TOUGH2 or MARNIE input files				
`-mode` **`clean`**	Cleans up a source code directory				
Input files					
`-in` *`X`*	In mode `run`, *`X`* is the TOUGH2 or MARNIE input file (relative or absolute path). In all other SITA modes, *`X`* is a test case file or a test case list (use relative paths starting from either the current directory or `$T2TESTS` or `$MARNIETESTS`). If *`X`* is set to `all`, every test case in the current directory and in `$T2TESTS` or `$MARNIETESTS` will be executed.		M	M	M
`-mob` *`X`*	*(MARNIE only)* Sets the mobilisation file (relative or absolute path)			M	
`-febe` *`X`*	*(MARNIE only)* Sets the FEBE file (relative or absolute path)			M	
`-out` *`X`*	*(MARNIE only)* Name of the output file. If not specified, the input filename is used and extended with "_out".			O	
Source code					
`-path` *`X`* (see also `-sc`)	Path of the source code directory. Aliases for path names are stored in `$T2GRS/aliases.json` and `$MARNIEPATH/aliases.json.` Except for mode `run,` the path option may pass multiple paths, like e.g. `-path "tough2/trunk  tough2/branches/user1".` In mode `clean`, `-path all` will clean all directories for which aliases have been specified.	M*	M*	M*	M*
`-eos` *`X`* (see also `-sc`)	*(TOUGH2 only)* EOS module of the executable, e. g. "eos7", "eos7r" (case insensitive).		M*	M*	M*
`-npmodule` *`Y`*	*(MARNIE only)* Maximum number of p-modules for the compilation of MARNIE. Allowed numbers: 2, 10, 150, 250, 400.			M*	M*

`-sc X`	This option specifies one or more source codes and can be used as an alternative to the `-path`/`-eos` and `-path`/`-npmodule` options for TOUGH2 and MARNIE, respectively. Examples: `-sc "tough2/trunk eos7 tough2/branches/nav eos7r"` `-sc "matrunk 400 matrunk 250"`.		O*	O*	O*
`-code A`	If the `-path X` directory has no version file, all subdirectories are scanned for a version file in order to find version `B` of code `A`.		O	O	O
`-version B`			O	O	O
Compiling					
`-clean` `-noclean`	`-clean` builds a new binary, whereas `-noclean` keeps old ones			O	O
`-compiler X`	Use Fortran compiler *X* (`gfortran` or `ifort`)			O	O
`-cf X`	Replace default compiler flags by flags *X*			O	O
`-addcf X`	Add flags *X* to the default compiler flags			O	O
`-cf95 X`	Same as `-cf` but for Fortran 90				
`-addcf95 X`	Same as `-addcf` but for Fortran 90				
`-compiler2 X`	Use also compiler *X* (doubles the number of executables). Setting this option will allow to specify the additional options `-c2f X`, `-addc2f X`, `-c2f95 X`, and `-addc2f95 X`				O
`-debug` `-nodebug`	Choose debugging options for compilation (only in `-mode test`)				O
Special testing options					
`-coverage` `-nocoverage`	Instructs SITA to generate code coverage information for the executable specified in the first entries of the `-path/-eos/-npmodule/-sc` options. SITA prints the coverage information to the HTML output and stores the coverage files in the folder `coverage`.				O
`-forcecalc`	Deletes pre-existing simulations results. By default, SITA keeps existing simulation results to avoid repetition of successful simulations in case of simulation failures.				O
`-htmlsilent`	Suppresses warnings in the HTML output				O
Queuing system (only available if `qsub` is installed on the system)					
`-qsub` `-noqsub`	Submits simulation jobs to the queuing system. SITA assumes the existence of a queuing system if the command `qsub` is available.			O	O
`-walltime X`	Walltime of the job(s) (DD:HH:MM:SS)			O	O
`-nodes X`	Number of nodes used by the job(s)			O	O
`-nprocs X`	Number of processors (on a node) used by the job(s)			O	O
`-q X`	Queue name, e. g. “dev” (development) or “prod” (production runs)			O	O
`-mail X`	Condition for email notification: Job has aborted (`a`), begun (`b`), or terminated (`e`)			O	O
`-mailto X`	Email address for notifications			O	O
`-name X`	Job name for the queuing system			O	O
`-wait X`	Checks job completion every *X* seconds (default is 10 seconds)				O
`-maxloop X`	Do not check job completion more than *X* times in succession (default is 60)				O
Other options					
`-silent` `-notsilent`	Suppresses console output from TOUGH2, MARNIE and *GNU make* (code compilation).			O	O

6.2 Performing code tests (mode `test`)

In order to perform code tests, the user has to define test cases. Test cases include input data for simulations and instructions that retrieve plot data (see chapter 8.6). In principle, test cases can be executed with different code versions. Since executables may require individual I/O formats SITA uses data interfaces to convert version-dependent data to version-independent data and vice versa.

If SITA is called in the test mode, this is called a *test run*. A test run can be performed with both, a single *test case* and a *test case list*. A test case list is a simple text file with the file name of one test case or test case list in each line (i.e. test case lists may be nested). Test case lists may include comments, which have to start with a hash key (#). Code coverage (the analysis of which code lines have been covered during code execution) is ascertained for the entire test run.

In the following examples, SITA is instructed to process the test case list `testcases.list`:

```
t2sita.pl -mode test -path tough2/trunk -eos eos7 -in testcases.list
masita.pl -mode test -path marnie/trunk -npmodule 250 -in testcases.list
```

The executable is specified by the options `-path`, `-eos`, and `-npmodule`. Note that the path depends on the local folder structure (see chapter 3).

If the environment variables `$PATH`, `$T2TEST`, and `$MARNIEPATH` have been configured properly (see chapter 4.2), SITA can be called from every directory and will place its output in the local directory. It is recommended to call SITA either from an empty directory or from the folder holding the test case files.

SITA generates a separate folder for each test case of the test run. The folders are named according to the path name that has been specified in the test case. Inside each test case folder, separate folders are created for each executable that has been invoked by the test case.

SITA builds executables using the Fortran compilers `gfortran` (default) or `ifort` unless the `-noclean` option is set. The compilation process can be influenced by several SITA options, which are listed in Tab. 6.1. It is possible to compare the results of executables that have been built with different compilers by setting the option `-compiler2 X`. This option triggers a second set of simulations using executables compiled with compiler `X`.

SITA proceeds with the simulations after the executables have been built. A test case can trigger more than one simulation. This is the case if

- more than one source code is used (`-sc` option),

- a second compiler is used (`-compiler2` option), or
- additional executables have been specified in the test case file.

If SITA finds that there already are simulation results in a simulation folder, it will not repeat these simulations (this can be suppressed by the `-forcecalc` option). The intention is to save simulation time if only a few simulations have to be repeated or if just the plots need to be changed. If existing simulations are kept, SITA prints a warning message, which can be suppressed by the `-htmlsilent` option.

If the system command `qsub` is available, SITA assumes the presence of a queuing system. The GRS-HPC-System uses `torque` and `moab` for job management and submission. Therefore, all queuing commands are specified for this system. If a test run triggers more than one simulation, each simulation will be executed as a separate simulation job. SITA will wait for the termination of all simulation jobs before it proceeds to analyse the simulation results (see options `-wait` and `-maxloop`).

Eventually, the HTML file `sita.html` is created in the current directory holding information on the test run, the test cases, the simulation results and, if requested, on the code coverage that has been achieved. Code coverage is only determined for the first executable that has been specified by the `-path`, `-eos`, `-npmodule`, or `-sc` options. SITA stores the coverage files in a folder named `coverage` inside the current directory.

6.3 Running a single simulation (mode `run`)

In mode `run`, SITA initiates a TOUGH2 or MARNIE run by calling the corresponding executables that have been built in the local copy of the code repository. The mandatory options are the same as for the `test` mode:

```
t2sita.pl -mode run -path tough2/trunk -eos eos7 -in INFILE

masita.pl   -mode   run   -sc   "matrunk  400"   -in   INFILE   -mob  MOBFILE
          -febe FEBEFILE
```

The `-in` option requires a regular TOUGH2 or MARNIE input file (instead of a SITA test case file or a test case list). Consequently, only a single simulation can be started in mode `run`. Executables are built in the same way as in mode `test,` except for the fact that the compiler's debugging option is not set (see chapter 7 for more debugging informations). The `-debug` option overrides this default setting.

If a queuing system is available, SITA will make use of it unless the option `-noqsub` has been chosen. The options `-walltime X`, `-nodes X` and `-nprocs X` are mandatory in this case. The simulation is executed in the current directory, i.e. the directory from which SITA was called. Both TOUGH2 and MARNIE will place their output there. In contrast to mode `test`, no folders are created.

6.4 Converting test case files

Test case files include a description of the simulation input in the JSON data format (see chapter 8 for more details). In the `convert` mode, SITA converts these input data to the regular input format of TOUGH2 or MARNIE.

The input data of a simulation is source code specific. SITA therefore needs information on the specific input format. This information is stored in the *input data interface* (s. chapter 9.1), which is located in the source code directory. When selecting a source code by means of the options `-path` and `-eos/-npmodule` options, the corresponding interface is selected automatically. .

The command

```
t2sita.pl -mode convert -path tough2/trunk -eos eos7 -in testcases.list
```

for the TOUGH2 code instructs SITA to convert all test cases listed in `testcases.list` to input files suitable for the executables specified by the `-path` and `-eos` options. SITA does not overwrite existing TOUGH2 input files. If multiple files are converted, a number will be appended to the generated file names.

6.5 Cleaning up the source code directory

The temporary files generated in the source code directory by the compilation process and the code coverage analysis can be erased by calling SITA in the `clean` mode:

```
t2sita.pl -mode clean -path tough2/trunk
masita.pl -mode clean -path marnie/trunk
```

The source code, the version file, and the data interfaces will be kept. If `-path all` is specified, all folders for which aliases have been specified in the `aliases.json` file (located in the SVN root folder) will be cleaned.

7 Building executables

SITA uses the programme *GNU make* /GNU 14/ to build executables from the source code. Therefore, every source code directory must contain a `makefile`. Executables can also be built without SITA by switching to the source code directory and running the `make` command. SITA includes makefiles for TOUGH2, TOUGH2-GRS, and MARNIE. The commands of these makefiles will be described now.

In order to build a TOUGH2 executable, `make` is called with the executable name

```
make executable_name
```

which generates the executable `executable_name` in the source directory. `make help` lists the available executables and options for the `makefile`. All executables are built at once if `make` or `make all` is executed. Note that the executable root names, which are created by the makefile, have to correspond to the executable names that are listed in the `bin` entries of the version files (see section 9.1), because these have to inform SITA on the executable names.

For MARNIE, *make* has to be called with the maximum number of p-modules:

```
make PMODULE=XXX
```

Presently, this number is restricted to values of `XXX =` 2, 10, 150, 250 and 400. If called without argument, SITA uses `PMODULE=400`.

The makefile offers commands that tidy up the source code directory:

- `make clean` deletes temporary files that have been created during previous compilations
- `make cleanexe` removes all executables
- `make cleangcov` and `make cleanicov` delete code coverage information for the gfortran compiler and the intel compiler, respectively.
- `make cleancov` deletes code coverage information for both compilers, gfortran and intel.

The following options are used to select the Fortran compiler and compiler options.

- `COMPILER=gfortran` or `ifort` chooses the Fortran compiler
- `COVERAGE=YES` sets the compiler flags for determining the code coverage
- `DEBUG=YES` sets the compiler's traceback option
- `FLAGS=""` overrides the default compiler flags
- `ADDFLAGS=""` adds compiler options
- `FLAGS95=""` overrides the default compiler flags for Fortran 90/95
- `ADDFLAGS95=""` adds compiler options for Fortran 90/95
- `COMPID=XY` adds a compiler ID to the executable name

Each option appends an option identifier to the executable name, which, at maximum, can have the following form:

```
<codename><module>-debug-cov-openmp-<compilername><compilerid>-newflags-
addflags.exe
```

If executables are built, the makefiles call the Perl script `writestampf.pl`, which creates the Fortran file `stamp.f90`. This Fortran file is included in the TOUGH2-GRS and MARNIE source code and prints information on the compiler, compiler options, compile time, source files, and file revision numbers to the simulation output.

8 Test cases

8.1 Introduction and terminology

Every execution of SITA in the `test` mode is called a *test run*. Test runs may cover multiple test cases. Every test run creates a single HTML output file. Code coverage is determined only for test runs.

This chapter focuses on test cases. A test case is defined for a certain physical problem. If only one executable is tested, the execution of a test case with SITA triggers exactly one simulation. The test case holds the input data for this simulation but it also contains instructions on which data is retrieved from the simulation output and how it is processed and displayed (see chapter 8.6). The instructions of a test case that display certain aspects of the simulation output (e.g. in form of a plot) are called an *analysis*. A test case usually contains several analyses for the same simulation.

The SITA mode that runs test cases is the `test` mode. The `-in` option passes the name of the test case file, which uses the JSON data format:

```
t2sita.pl -mode test -path tough2/trunk -eos eos7 -in testcase.json
masita.pl -mode test -path marnie/trunk -npmodule 250 -in testcase.json
```

The `-in` option can also pass a test case list, which is a text file containing names of test cases or other test case lists, which are separated by line feeds. The `-in` option requires a path that is defined relative to the current directory or to `$T2TEST` or `$MARNIETESTS`. Using `-in all` executes every test case that is located in the current directory and in the folder paths `$T2TEST` or `$MARNIETESTS`.

Test case files may be supplemented by additional files e. g. for initial conditions or mesh data. There are instructions in the test case file which copy these files to the simulation directories. Before we turn to these instructions, we will first take a look at the high-level structure of test case files.

8.2 High-level structure of a test case file

Test case files are written in the JSON data format (see attachment A or /JSON 13/). The reader should be familiar with this format in order to understand the following examples. Name/value pairs of JSON will be called “fields” in the following.

Fig. 8.1 shows the high-level structure of a test case file. There are four fields on this level: the `author`, `test case`, `analyses`, and `input` (or `input-marnie`) fields. There are two additional fields `solute` and `chemical` for the code TOUGHREACT. The fields may have arbitrary order because a JSON object `{…}` is an unordered collection of fields.

```
{
    "authors" : [
       {"name" : "my name", "email" : "myself@home″ },
…
    ],

    "test case" : {
…
    },

    "analyses": [
…
    ],


    "input" : {
...
    }
}
```

Fig. 8.1 High-level structure of a test case file

The `test case` field includes general information on the test case (chapter 8.5). Instructions for analysing the simulation results are part of the `analyses` field (chapter 8.6). The input data for TOUGH2 simulations is provided by the `input` field, those of MARNIE simulations in the `input-marnie` field (chapter 8.7). Test cases may be used to generate plots without running simulations. In this case, the `input` field may be omitted.

8.3 Description fields

The `test case` and `analyses` fields may contain `description` subfields like

```
"description" : "This is a very special test case",
```

Description fields may include HTML code for text markups. For example, it is possible to structure the description by using line breaks (`<br>`) or paragraphs (`<p>…</p>`). SITA also allows the insertion of line breaks by passing an array of lines to the `description` field:

```
"description" : ["line 1,",
                 "line 2,",
                 "and line 3"],
```

This method increases the readability of the description field.

SITA descriptions fields accept **La**mport **TeX** (LaTex) /LAT 16/ formulas (in both, *inline* and *displayed math* mode including the commands of the `amsmath` and `siunitx` packages).

SITA offers the following commands to introduce LaTeX formulas:

- `\L` and `\l` embraces general LaTex commands.
- `\[` and `\]` embraces displayed formulas (formula appears in a separate line)
- `$` and `$` embraces in-line formulas.

If LaTeX code is used in description fields, the JSON format requires that backslashes (\) and quotes (") are masked by backslashes. For example

```
\[ \frac{x}{2}=y \] for $t=0$
```

has to be inserted as

```
"description" : "\\[ \\frac{x}{2}=y \\] for $t=0$ ",
```

Description fields can retrieve parameter values from the `input` field (input fields are described in chapter 8.7). The respective Perl reference to this parameter value has to be enclosed by double hash keys:

```
"description" : "NOITE has the value ##{PARAM}{NOITE}## and
Material 1 has the name ##{ROCKS}{MATERIAL}[0]{MAT}##",
```

8.4 EOS selection lists for TOUGH2 test cases

If TOUGH2 test cases are compatible with different EOS modules, it is possible to compare the results achieved by these EOS modules. A problem is that different EOS modules usually require different input data. Test cases may therefore contain *EOS selection lists* which are EOS specific parts of a test case.

Presently, the use of EOS selection lists is restricted to the `input` and the `test case` fields. Note that the way EOS selection lists are used differs between the `input` and `test case` field. Future SITA versions will probably introduce a more general way of defining EOS selection lists in test cases.

In `input` fields EOS selection lists have the form

```
                {"EOS1" : …, "EOS2": … , …}
```

which means that they can be inserted as field values (including the value of the `input` field itself). In the following example of an `INDOM` data block the `MATERIAL` field is substituted by a EOS selection list (marked in boldface).

```
         "INDOM": {
             "KEYWORD": "INDOM----1----*----2----*----3----*----4----*----
5----*----6----*----7----*----8",
             "MATERIAL":
               {
                 "EOS7" :
                 [
                   {
                       "MAT": "MAT1",
                       "X" : [1.0E+5, 0,  0.0, 5.0]
                   },
                   {
                       "MAT": "MAT2",
                       "X" : [1.0E+5, 0,  0.0, 5.0]
                   },
…
                 ],
                 "EOS9" :
                 [
                   {
                       "MAT": "MAT1",
                       "X" : [1.0E+5]
                   },
                   {
                       "MAT": "MAT2",
                       "X" : [1.0E+5]
                   },
…
                 ]
               }
```

```
        },
```

In `test case` fields (see chapter 8.5), EOS selection lists have the form

```
              "EOS1" : {…}, "EOS2" : {}, …
```

Here is an example:

```
     "test case" : {
         "title" : "sample test",
         "version" : "1.0",
         "grid": "mytest.mesh",
         "EOS7" : {
              "initial conditions": "eos7.incon",
              "description" : "my description for eos7"
         },
         "EOS7R" : {
              "initial conditions": "eos7r.incon",
              "description" : "my description for eos7r"
         },
         "makedir" : "mytest",
         "suitability": [ ["TOUGH2","EOS7"], ["TOUGH2","EOS7R"] ]
     },
```

The only field that cannot be substituted by an EOS selection lists is the `suitability` field, which defines the range of application of the test case.

8.5 General test case data

Here is an example of a `test case` field for the TOUGH2 code. The meaning of the individual fields will be explained in the following.

```
         "title" : "",
         "description" : "",
         "version" : "1.0",
         "makedir" : "mytestcase",
         "input files": [
             "copy_this_file_without_renaming_it.dat",
             ["mytestcase.mesh","MESH"],
             ["initialconditions.incon","INCON"]
         ],
         "suitability": [
             ["TOUGH2-GRS","EOS7", ">=0.1.a"]
         ],

         "walltime" : "00:04",
         "nodes" : "1",
         "nprocs" : "2",
         "query period" : "5",
         "maxloop" : "5",
         "wait" : "5"
```

The `title` field

Holds the title of the test case. This title will be displayed in the output.

The `version` field

The version number of the test case. This field has no effect on the test run.

The `makedir` field: How simulation results are stored

SITA creates the output file `sita.html` in the current directory (the directory from which SITA was called).

In the current directory, a separate directory is created for every test case in order to store the simulation results. These *test case folders* are named according to the values of the test cases' `makedir` fields, which are mandatory.

Test case folders split up into *simulation folders* for each executable. The folder names indicate which code, code version, module, compiler, and compiler options have been used.

The `input files` field: Copying and renaming files

The optional `input files` field instructs SITA to copy files from the test case folder to the simulation folders. TOUGH2 test cases are often supplemented by grid files and files containing initial conditions, which have to be copied to the simulation folders under the name `MESH` and `INCON`, respectively. When adding a file name to the `input files` field, the file is copied. When adding an array instead, like `["mytestcase.mesh","MESH"]`, the file `mytestcase.mesh` is renamed to `MESH`. See the following example:

```
        "input files" : [
                         ["file1_will_be_copied_and_renamed","new_name"],
                         ["file2_will_keep_its_name",""],
                          "file3_will_keep_its_name_too",
                       ]
```

In replacement of the `input files` field, users of TOUGH2 can use the fields `grid` and `initial conditions` to create the files `MESH` and `INCON`:

```
        "grid": " mytestcase.mesh",
```

MARNIE users use the `mob` and `febe` fields instead (no renaming).

```
  "febe": "myfebefile",
```

The `suitability` field: Scope of the test case

Test cases can only be used for a restricted range of code versions or modules (EOS modules for TOUGH2 or p-modules for MARNIE), which is specified by the `suitability` field. This field may contain multiple entries of the form `["code_name", "module_name"]` or `["code_name", "module_name", "version_range"]`, which indicate the permitted code and code version. Version ranges may be specified by version numbers and operators (e.g. "`>= 1.1`" or "`<9.0`"). A lowercase letter indicating the patch version will be ignored. SITA determines the code and code version from the `version.json` of a source code (see chapter 9.1) in order to decide whether the test case can be executed.

Queuing system (`walltime`, `nodes`, `nprocs`, `query period`, `q` and `maxloop` fields)

The way simulations are executed on a cluster can be controlled by special instructions inside the test case. For this purpose, the `test case` field can be supplemented by the fields `walltime` (walltime in format DD:HH:MM:SS), `nodes` (number of nodes), `nprocs` (number of processors), `q` (queue in which the job is running, e. g. `dev` or `prod`), `query period` (period in seconds after which job status is checked), and `maxloop` (number of status checks).

8.6 Analyses: retrieving, processing and displaying data

Analyses instructions are provided by the `analyses` field of the test case. Analyses for MARNIE are not yet fully developed. For this reason most of the analyses types, which are being presented below, only apply to the TOUGH2 code.

The analysis type is selected by the `type` field. The following types are available:

- `"type": "scalar"` retrieves a parameter value from the simulation output and prints it. It is necessary to specify a grid location and a simulation time. (TOUGH2 only)
- `"type": "mass"` prints the total component masses at two simulation times and calculates the absolute and relative mass change. (TOUGH2 only)
- `"type": "time series"` plots a time series for a specified parameter, location (or location array), and time range. (TOUGH2 only)
- `"type": "profile"` plots the value of a parameter for selected elements along a given scan line. (TOUGH2 only)
- `"type": "datafiles"` plots xy data files only.
- `"type": "time steps"` plots the evolution of the time step width.
- `"type": “readplot”` uses the code *readplot* to extract the data from MARNIE output files. (MARNIE only)

```
The following fields, which will be explained in the following subchap-
ters, can be part of every analysis.{
    "title": "",              plot title
    "remark": "",             optional remark
    "type": "",               type of analysis


    "save": "my file name",         save analysis results for inclusion in
                                    other analyses (optional)

    "mode": "silent" ,              don't create any HTML output for this
                                    analysis (optional) and do not count this analysis

    "limit datapoints to": 10000,         reduces number of data points that are
                                          retrieved from the simulation output


    "simulations": [       trigger and compare with the following source code: (optional)
        {
            "path": "tough2-grs/trunk",
            "code": "tough2-grs",
            "version": "01",
            "eos": "eos7"
        }
    ],

    "functions": [            functions to plot (optional)
        [ "x > -5 ? 1 : 0" , "my analytical solution" ]
    ],

    "xy-data": [              xy-plots (optional)
        [ "xy-data.txt" , "line title"],       and/or
        [ "xy-data.txt" , "line title" , "1:3", "x1y2"]
    ],


    "gnuplot files": [        files containing gnuplot commands (optional)
```

```
          [ "filename1" ],
          [ "filename2" ]
     ],

     "gnuplot commands": "", gnuplot commands (optional)

     "plot options": " "       SITA specific plot commands (optional)

},
```

8.6.1 Plotting functions and xy-data

SITA can combine simulation results, xy data files, and analytical functions in a single plot. Plots are created with *gnuplot* and *ghostscript /GHO 14/, /WIL 16/* and are embedded in SITA's HTML output.

xy data

The `xy-data` field is used to plot xy data (whitespace or comma separators). The data files have to be located either in the test case folder or in the subfolder `data`.

The `xy-data` field specifies an array of plot specifications. A plot specification is an array containing a file name, a line title (will appear in the plot key), an optional column selector (`col1:col2`), and an optional axis selector (`x1`, `x2`, `y1`, `y2`):

```
          [ "xy-data.txt" , "line title" , "1:3", "x1y2"]
```

Without column selector, the first two columns are used as x and y values. Without axis selector `x1y1` is used.

Data files created by means of the `save` field hold additional information on the executable time of printout and data source. This information is included in the line title by default. In order to plot only the line title without the additional information, the title specification has to be preceded by a hash key:

```
          [ "xy-data.txt" , "#line title" , "1:3", "x1y2"]
```

If the `-sc` option of SITA specifies more than one executable, the `save` command creates one file for each executable. For example,

```
        "save": "my-file.txt"
```

generates the files `my-file.txt` and `my-file-2.txt` for the first and second executable listed by the `-sc` option, respectively. By default,

```
        "xy-data": [[ "my-file.txt" , "line title"]]
```

checks if the files `my-file-`*i*`.txt` exist and plots them, too. This default behaviour can be supressed by preceding the file name by a hash sign (#).

```
        "xy-data": [[ "#my-file.txt" , "line title"]]
```

This will only plot the file `my-file.txt`.

Functions

Analytical functions can be plotted by means of the `functions` field, which is an array of function specifications. Each function specification holds a mathematical expression in *gnuplot* syntax and a function title:

```
                "functions": [
                    [ "4*sqrt(x)" , "analytical solution" ]
                ]
```

Because function definitions can grow very long, SITA offers two possibilities of providing these definitions:

- The first possibility is to use the field `gnuplot commands`. The contents of this field are read by *gnuplot* prior to the execution of the function.

```
                "gnuplot commands": "a=4; f(x)=a*sqrt(x);",
                "functions": [
                    [ "f(x)" , "analytical solution" ]
                ]
```

  Note that the `gnuplot commands` field asks for *gnuplot* commands, only. Since there is only one line for all commands (separated by semicolons) readability suffers from long function definitions.

- In order to avoid this problem, the function definition can be shifted to a text file (line breaks are permitted here). This text file can be imported by the `gnuplot files` field:

```
                "gnuplot files": [
                    "file_defining_function_f(x).txt"
                ],
                "functions": [
                    [ "f(x)" , "analytical solution" ]
                ]
```

  This is particularly useful if the same function definitions are used in several analyses or test cases. Note that the `gnuplot files` field is processed before the `gnuplot commands` field. SITA requires that the file paths of the `gnuplot files` field are specified in relation to the test case directory.

Customizing the plot

Plots can be customized either by adding gnuplot commands to the `gnuplot commands` field or by using the `plot options` field. The commands of the `gnuplot commands` field are executed after that of the `plot options` field.

The *gnuplot* commands that can be used to customize the plot are described in /GNU 14/. Here is a list of commands that are frequently used:

- `set format <axes> '<format>'` sets the format of the tic-mark labels. For example, `set format x '%.2f'` uses floats with two decimal places for the x-axis /GNU 14/. The axis can be specified as `x`, `y`, `x2`, or `y2`.
- `set <axis>label '<text>'` sets the title of an axis.
- `set <axis>range [min:max]` sets the plot range for an axis.
- `set logscale <axes> <base>` enables log-scaling of axes. `base` is the base of the logarithm (default is 10), e.g. `set logscale xy 10` for double logarithmic scaling.
- `set grid no<axis>tics|<axis>tics nom<axis>tics|m<axis>tics` enables grid lines for major and minor tic marks, e.g. `set grid xtics ytics mxtics mytics` shows the grid for all major and minor tic marks.
- `set m<axis>tics <number>` sets number of minor tic sub-intervals between major tics, e.g. `set mxtics 2`. Especially useful for logarithmic scaling of axes.

- `set <axis>tics mirror|nomirror in|out rotate|norotate` controls the tic attributes. `mirror` displays tics also at the opposite border of the plot. `in|out` chooses between in- and outwards tics. `rotate` rotates the tic lables by 90 degrees.
- `set tics scale <major>,<minor>`: controls the size of the tic marks. The first value controls the size of major tics, the second that of minor tics `(<major>` defaults to 1.0, and `<minor>` to <major>/2).
- `set key on|off inside|outside left|right|center top|bottom|center vertical|horizontal width <width_increment> height <height_increment>` adds a key to the graph. Positions indicate the position of the key. The height and width of the key can be increased or decreased. SITA's default setting is `set key on outside center bottom horizontal width 0 height 0`. See /GNU 14/ for more details.
- `set mapping cartesian|spherical|cylindrical` chooses the coordinate system (default is cartesian).
- `set style data lines` uses dashed lines (default).
- `set style data linespoints` uses dashed lines with markers.
- `set style data points` uses markers only.

If the user does not want to use *gnuplot* commands, he can also customize the plot using the `plot options` field, which contains the following SITA specific commands (separate commands by blanks):

- `-xtitle 'text'` sets the title of the x-axis
- `-ytitle 'text'` sets the title of the y-axis
- `-xrange [min:max]` defines a plot range for the x-axis
- `-yrange [min:max]` defines a plot range for the y-axis
- `-x2title 'text'` sets the title of the secondary x-axis
- `-y2title 'text'` sets the title of the secondary y-axis
- `-x2range [min:max]` defines a plot range for the secondary x-axis
- `-y2range [min:max]` defines a plot range for the secondary y-axis

- `-xlog` sets a logarithmic x-axis
- `-x2log` sets a logarithmic secondary x-axis
- `-ylog` sets a logarithmic y-axis
- `-y2log` sets a logarithmic secondary y-axis
- `-lwdiag X` sets the line width to `X` in the `set terminal` command
- `-nokey` suppresses the key
- `-key X` creates the *gnuplot* command "`set key X`"
- `-keyspacing X` sets the vertical key spacing to `X`
- `-width X` sets the width of the postscript terminal in pixel if the resolution was 300 dpi (default: 1500 px)
- `-height X` sets the height of the postscript terminal in pixel if the resolution was 300 dpi (default: 835 px)
- `-resolution X` (or `-res X`) sets the resolution for the conversion from postscript to the png format (default: 300 dpi)
- `-shortlinetitle` adds only the code and module information to the line title.

The following gnoplot options are used as default.

```
set style data lines
set size 1,0.8
set format y '%G'
set format x '%G'
set bmargin at screen 0.3
set tics in
set mxtics 10
set mytics 10
set border back lw 0.3
set nozeroaxis
set grid lc rgb '#888888' lt 1 lw 0.1
set key outside below box lw 0.5 lc rgb '#888888' spacing
set key reverse Left
set key left
set key font 'Arial,12'
set autoscale xy
```

8.6.2 Comparing simulation results

Simulation results can be compared to results of other simulations. A way of doing this is to add a list of additional executables by means of the `simulations` field. All simulation results will be combined in the printout or plot. Another way to compare executables is to use the SITA options `-sc` or `-compiler2` (see section 6.1), which leave the decision on the executables to the SITA command (i.e. it is not hard-wired in the test case).

If a TOUGH2 test case compares executables that use different EOS modules, the test case has to use EOS selection lists in order to be compatible to all executables (see section 8.4).

SITA cannot execute simulations for different physical problems at the same time because these have to be defined in separate test cases. However, there is an indirect way of doing this by storing the plot data of an analysis into a file using the `save` field in the analysis definition. For every executable the command

```
"save": "plot_data_of_this_analysis.txt",
```

generates a data file in the `data` folder, which is located in the current directory. Each file receives an executable-specific name. The file name for the first executable (which is specified by the `-path` and `-eos`/`p-module` or `-sc` option) is the value of the `save` field. For the following executables this value is extended by a number (e.g. `plot_data_of_this_analysis-2.txt`).

The files generated by the `save` field can be included by other analyses or test cases by means of the `xy-data` field, which holds the desired file name (see chapter 8.6.1). SITA searches for this file in the test case directory and `data` folder (in this order). Unique file names for storing the plot data are mandatory.

If the plot data shall be stored but not displayed, the entire analysis can be hidden by

```
"mode": "silent"
```

The skipped analysis will be ignored in the numbering of the analyses.

8.6.3 TOUGH2 specific features

8.6.3.1 Data sources

Analyses usually execute simulations and retrieve output parameters. Output parameters have to be specified with regard to location and time. The format of location and time information depends on the data source.

Data sources of TOUGH2 are the main output (directed to the standard output channel `STDOUT`) and the files `FOFT`, `COFT`, and `GOFT`. TOUGH2-GRS also creates a `DOFT` file for domain specific properties (TOUGH2 domains are also called *materials* or *rocks*). The files `ele_main` and `con_main` have been introduced to TOUGH2-GRS to store element and connection specific parameters of the main printout in the format of the `FOFT`, `COFT`, and `GOFT` files. These files have also been described in chapter 5.2.

The data source has to be specified by the field `source` like in the following example.

```
                "source": "FOFT"
```

SITA knows which data sources are available, because these are registered in the output interface, which is explained in chapter 9.3. The data sources `"stdout"`, `"foft"`, `"coft"`, and `"goft"` are available for every TOUGH2 based code. TOUGH2-GRS interfaces also introduce the sources `"doft", "ele_main"`, and `"con_main".`

Some analyses operate exclusively with the `"STDOUT"` source. In this case, the source specification can be omitted.

8.6.3.2 Specifying times

Time values for time series data are specified by a numerical value and a time unit. Currently available units are

- second: “sec”, “s”
- minutes: “min”
- hours: “h”

- day: “d”, “day”, “days”
- julian year (365.25 days): “a”, “year”, “years”, “julian years”
- siderial year (365.256363004 days): “siderial years”
- tropical year (365.24219): “tropical years”
- gregorian year (365.2425 days): “gregorian years”

For all data sources, time values can also be specified by source specific printout numbers because each printout is associated with a time value. Which time value is assigned to which printout depends on the simulation but also on the code. For example, some TOUGH2 based codes make printouts at iteration 0 whereas others don’t.

Printouts are numbered by order of appearance in the source file. Possible printout specifications are `"first printout"`, `"last printout"`, and `"printout` *(number)*`"`.

8.6.3.3 Specifying locations

TOUGH2 elements, connections and materials are referred to as *locations*. A location specification has the form `"A11  1"` (for elements), `"A11  1A21  1"` (for connections), or `"mat  2"` (for materials).

For all data sources, the location nam `"sum"` will sum up the parameter values of all locations that are present in the data source.

If the data source `DOFT` is used, the location `"active elements"` will retrieve parameter values for all active elements.

Multiple locations are specified by an array construction:

```
"location": ["A11 1", "B10 1", "A11 3", …],
```

8.6.3.4 Specifying parameters

Output parameters are specified by the `par` field[1]:

```
                 "par": "Sliq",
```

The available parameter names are defined in the output interface file (see chapter 9.3). If not documented elsewhere, the user can open this file to lookup the available names. Note that SITA does not use the parameter names used in the output to `STDOUT`, because these names might vary between different code versions.

For time series and profile plots (see sections 0 and 8.6.3.8), module specific parameters can make use of EOS selection lists:

```
                 "par" : {
                    "EOS7": "mypar1",
                    "EOS5": "mypar2"
                 }
```

8.6.3.5 Printing mass changes

TOUGH2 printouts usually include information on the total component masses inside active elements. SITA can compare the component masses at two different printout times by means of the analysis type `mass`:

```
                 "type": "mass",
                 "time 1": "printout 1",
                 "time 2": "printout 2",
```

The data source for this analysis is always `"STDOUT"` and therefore does not have to be defined.

1 It replaces the equivalent field “`indicator`” of former SITA versions.

8.6.3.6 Printing parameter values

Analyses of type `scalar` generate a printout of parameter values for a certain grid location and TOUGH2 printout.

```
                "type": "scalar",
                "source": "ele_main",
                "time": "first printout",
                "criterion": "x < 0.10001 and x > 0.0999",
                "location": "A11 1A21 1",
                "par": "FLOF",
```

The retrieved parameter value can be checked by defining a criterion in the `criterion` field (Perl syntax) using the variable "x" for the parameter. If the criteria are not met, an error message is written to `sita.html`.

8.6.3.7 Plotting a profile

The analysis type `"profile"` is used to generate profile plots:

```
                "type": "profile",
                "xrange": [0,1],
                "yrange": [0,1],
                "zrange": [-20,0.0],
                "scanline direction": "z",
                "time": "printout 2",        or "time": [1.0E3,8.67E4],
                "par": "Sgas",
                "x-axis": "2*$p-1",
                "y-axis": "$f+10",
```

Only those grid elements are included in the plot whose centres lie inside the range specified by the `xrange`, `yrange`, and `zrange` fields. The `scanline direction` field tells SITA which coordinate component to use for the horizontal axis of the plot (permitted values are "`x`", "`y`", and "`z`").

A post-processing of the parameter values and scanline positions can be achieved by the `x-axis` and `y-axis` fields. These fields hold Perl-style formulas, which make use of the following variables.

- `$p`: position
- `$t`: time
- `$dt`: time step width

- `$f[0]`, `$f[1]`, `$f[2]` , … : parameter values, in the order of parameter array passed by the `par` field. `$f[0]` is the first parameter. `$f` is identical to `$f[0]`.
- `$Sfdt` =`$Sfdt[0]`, `$Sfdt[1]`, `$Sfdt[2]` , … : time integration of `$f[i]`. If `$f` is a rate, it should refer to the same time unit as `$t` for a correct accumulation of `$f`.

scanline positions `$p` and parameter values `$f` (see the above example).

A printout time has to be specified either by a printout number or by a time value in seconds, which has to equal the exact time value of the printout. Parameter names are specified by the `par` field. Multiple parameters can be passed by an array of parameters:

```
                "par": ["Sgas", "Sliq"],
```

The `source` field has to be set either to `"STDOUT"` or to `"ELE_MAIN"`. If the `source` field is omitted, `"ELE_MAIN"` will be used as data source.

8.6.3.8 Plotting time series

Time series are extracted from the `FOFT`, `COFT`, `GOFT`, `DOFT`, `ELE_MAIN`, and `CON_MAIN` files. Therefore, the user has to set the `source` field to the desired data source:

```
                "type": "time series",
                "source": "FOFT",
                "from time": "1 sec",
                "to time": "1E+9 sec",
                "time unit": "sec",
                "limit datapoints to": 10000,
                "location": ["A11 1"],
                "par": ["Sgas","Pgas"],
                "x-axis": "$t",
                "y-axis": "$f",
```

If no source is specified in the `source` field, `ELE_MAIN` is used. The time range for data extraction is defined by the `from time` and `to time` fields. If SITA plots time series of executables which produce output with different time units, it will convert the time information to seconds unless the user prescribes another time unit using the `time unit` field.

Time series with a very large amount of data points can slow down the execution of SITA significantly. The `limit datapoints to` field allows reducing the number of data points to the value specified by the field. The spacing of data points considers whether a linear or a log scale is used on the time axis.

The available plot parameters (values given in the `par` field) are determined by the output interface except for the parameter "`time step`" which returns the time step number and does not require the declaration of a location:

```
                "par": "time step",
```

If multiple parameters or locations are given (using parameter or location arrays, respectively), SITA will superimpose the corresponding time series in a single plot. However, only a single line is plotted if the `x-axis` or `y-axis` field is used, because SITA assumes that the user wants to combine the listed parameters in a single mathematical expression.

The `x-axis` and `y-axis` hold Perl-style formulas, which use the following variables.

- `$t`: time
- `$dt`: time step width
- `$f[0]`, `$f[1]`, `$f[2]` , … : parameter values, in the order of parameter array passed by the `par` field. `$f[0]` is the first parameter. `$f` is identical to `$f[0]`.
- `$Sfdt` =`$Sfdt[0]`, `$Sfdt[1]`, `$Sfdt[2]` , … : time integration of `$f[i]`. If `$f` is a rate, it should refer to the same time unit as `$t` for a correct accumulation of `$f`.

The variables can be combined to mathematical expressions (Perl syntax). In the following example the product `Kintr` x `Vphys/Vevol` is plotted against time.

```
                "par": ["Kintr","Vphs/Vevol"],
                "x-axis": "$t",
                "y-axis": "$f[0]*$f[1]",
```

8.6.3.9 Plotting the time step evolution

The evolution of time step sizes can be plotted with the `time steps` analysis:

```
                "type": "time steps",
```

The same plot can be created by means of the `time series` analysis (`type: "time series","x-axis": "$t", "y-axis": "$dt"`). However, there is a small difference between these two plotting methods. The `time steps` analysis will take its data from the TOUGH2 printout to the STDOUT channel whereas the `time series` analysis retrieves data from the time series files like `FOFT` or `COFT`, which may not include all time step printouts.

8.6.4 MARNIE specific features

The `readplot` analysis only applies to MARNIE. MARNIE's *readplot* programme reads selected output data from the binary file (*.pd) and writes it to an ASCII-output file, which can be used as input file e.g. for plot routines. Analyses of type `readplot` require the fields listed in Tab. 8.1 for data selection.

Tab. 8.1 Mandatory parameters for the *readplot* analysis

Field name	Field vValue
`varnam`	Array containing up to eight names of output variables. Possible variable names are listed in the *.out files generated by MARNIE (e.g. “VOLFLOW” for the volume flow, “PRESS” for the pressure, and “`CI`(*i*)” for the concentration of component *i*. If `nanzkomp` > 0, it is only allowed to specify one component-specific variable (like `CI`).
`nanzkomp`	If `nanzkomp` = 0 (default value), only those variables will be read that are specified by the `varnam` array. If `nanzkomp` > 0 and a component specific variable `X`(i) has been specified, the data values of variables `X`(i) to `X`(i+`nanzkomp`-1) will be read. Maximum value for `nanzkomp` is 50.
`tmoban`	Starting (simulation) time for data extraction in seconds.
`tmoben`	End (simulation) time for data extraction in seconds.
`wertmi`	Data values of the current timestep are only printed to the ASCII outputfile if at least one data value of one output variable is larger than `wertmi`.
`difant`	Data for the current timestep is only written to the ASCII outputfile if the relative value change is larger than `difant` for at least one of the selected variables.
`faktim`	Time is divided by `faktim` and variable values are multiplied by `faktim`. Only applicable for rate variables. Set `faktim =1` otherwise.
`nanzpip`	Specifies which information is given by the `pmodule` field. `nanzpip = 999`: The `pmodule` array specifies the first and last p-module that is printed. `nanzpip<=12`: The `pmodule` array specifies holds `nanzpip` p-module numbers.
`pmodul`	Array specifying the printed p-modules. The required box-numbers of these p-modules must be defined by `box1` and `boxn` (see below), defining a range of box numbers.
`box1`	Number of the first p-module box that printed. `box1=1, boxn=99` prints all boxes.

Field name	Field vValue
`boxn`	Number of the las p-module box that printed. `box1=1, boxn=99` prints all boxes.

Here is an example of a complete set of *readplot* parameters:

```
             "type" : "readplot",
             "nanzkomp" : 0,
             "varnam" : ["POROS"],
             "tmoban" : 0.0,
             "tmoben" : 3.1536E13,
             "wertmi" : 1e-10,
             "difant" : 1e-15,
             "faktim" : 1.0,
             "dztrnz" : "PUNKT",
             "artausg" : "DATEI",
             "nanzpip" : 999,
             "pmodul" : [1,10],
             "box1" : 1,
             "boxn" : 99,
```

The readplot analysis also accepts the x-axis and y-axis fields, which use the same Perl variables as described in chapter 8.6.3.8. Example:

```
                 "x-axis": "$t",
                 "y-axis": "$f",
```

8.7 Simulation data

The input data for TOUGH2 and MARNIE simulations are stored in the `input` field and the `input-marnie` field, respectively. The content of these fields reflects the input syntax that is defined by the input interface (see chapter 9.1). If the user has to construct an `input` field without having a template at hand, he should study the input interface that is stored in the source code folder of the executable he uses. Details on how the `input` field is determined by the syntax description of the input interface will be given in chapter 9.1. However, there are some general aspects that apply to all input interfaces and shall therefore be introduced beforehand.

For each data block of the TOUGH2 or MARNIE input data, there is a separate data block entry

```
         "DATA BLOCK NAME": { … },
```

in the `input`/`input-marnie` field. For example, a hash for a TOUGH2 `MULTI` block can look like this:

```
        "MULTI": {
            "KEYWORD": "MULTI----1----*----2----*----3----*----4----*----
5----*----6----*----7----*----8",
            "NK" : 5,
            "NEQ" : 5,
            "NPH" : 2,
            "NB" : 6,
            "NKIN" : 0
        },
```

Data block entries can have arbitrary order in the `input`/`input-marnie` field. (The order in which they are printed to the code's input data file is determined by the input interface.)

For TOUGH2, the `input` field can make use of *EOS selection lists*

```
              {"EOS1" : …, ″EOS2‶, … , …}
```

which allows the construction of test cases that are suitable for different EOS modules. Here is a complete example of an `"INDOM"` data block, which uses an EOS selection list:

```
        "INDOM": {
            "KEYWORD": "INDOM----1----*----2----*----3----*----4----*----
5----*----6----*----7----*----8",
            "MATERIAL":
              {
                "EOS7" :
                [
                  {
                      "MAT": "MAT1",
                      "X" : [1.0E+5, 0,  0.0, 5.0]
                  },
                  {
                      "MAT": "MAT2",
                      "X" : [1.0E+5, 0,  0.0, 5.0]
                  },
…
                ],
                "EOS9" :
                [
                  {
                      "MAT": "MAT1",
                      "X" : [1.0E+5]
                  },
                  {
                      "MAT": "MAT2",
                      "X" : [1.0E+5]
                  },
…
                ]
              }
        },
```

Please note that the syntax of the EOS selection list is not part of the syntax description that is stored in by the input interface (instead it is hard-wired into the SITA code).

Input files of TOUGH2 and MARNIE may contain recurring data blocks with similar meaning like e.g. a repeated definition of material properties (see chapter 5.2). The user can provide such data blocks by means of *repetition arrays* (an array is delimited by square brackets; for more details see attachment A). Repetition arrays are composed of elements with identical format and syntax which are processed according to their order in the array. The following example shows the repetition array that has been used in the above `INDOM` example:

```
                    [
                      {
                          "MAT": "MAT1",
                          "X" : [1.0E+5, 0,  0.0, 5.0]
                      },
                      {
                          "MAT": "MAT2",
                          "X" : [1.0E+5, 0,  0.0, 5.0]
                      },
…
                    ],
```

9 Handling version-dependent source code

Test cases can be executed with different code versions. Since executables may require individual I/O formats, SITA uses data interfaces to convert version-dependent data to version-independent data and vice versa.

Every source code directory is accompanied by a JSON file that describes how the simulation data of a test case is converted into version-dependent TOUGH2 or MARNIE input. This JSON file is the *input interface*. Another JSON file makes the version-dependent output of TOUGH2 accessible to SITA. This file is called the *output interface*. Every source code directory has a JSON file `version.json` in which the file names of the input and output interfaces are registered (see chapter 3.3).

9.1 Version files

The version file and the interface files use the JSON data format (see attachment A or /JSON 13/). Here is an example for TOUGH2

```
{
    "code": "TOUGH2-GRS",
    "version" : "01a",
    "modules" : ["EOS7", "EOS7R"],
    "EOS7": {
        "bin": "t2eos7.exe",
        "in": "t2eos7eos7r-in.json",
        "out": "t2eos7-out.json"
    },
    "EOS7R": {
        "bin": "t2eos7r.exe",
        "in": "t2eos7eos7r-in.json",
        "out": "t2eos7r-out.json"
    }
}
```

and for MARNIE:

```
{
    "code": "MARNIE",
    "version" : "10.3",
    "modules" : [2,10,150,250,400],
    "2" : {
        "bin": "marnie.exe",
        "in": "marnie-in.json",
        "out": "marnie-out.json"
    },
    "10" : {
        "bin": "marnie.exe",
        "in": "marnie-in.json",
        "out": "marnie-out.json"
    },
    "150" : {
        "bin": "marnie.exe",
        "in": "marnie-in.json",
        "out": "marnie-out.json"
    },
    "250" : {
        "bin": "marnie.exe",
        "in": "marnie-in.json",
        "out": "marnie-out.json"
    },
    "400" : {
        "bin": "marnie.exe",
        "in":  "marnie-in.json",
        "out": "marnie-out.json"
    }
}
```

9.2 Input interfaces

Input interfaces are files in JSON format which describe the input syntax of a source code. The input interface is therefore located in the source code folder. In order to understand function of input interfaces, the reader should be familiar with the input syntax of the code he uses (TOUGH2 base code or MARNIE) and with JSON and Perl expressions.

MARNIE input files require input blocks like

```
------GENERAL DATA    -----
…
------NETWORK           ------
…
------PBLOCK            ------
…
------VBLOCK            ------
…
------P-MODUL-SECTIONS  ------
…
```

each of which starts with a keyword (e.g. `------PBLOCK`) and includes input values in fixed format. A detailed description of the MARNIE input is given in /FIS 02/.

Fig. 9.1 and Fig. 9.2 show examples of TOUGH2 and MARNIE input interfaces, respectively. An input interface is a JSON object containing the five fields `title`, `description`, `version`, `authors`, and `structure`. Since the meaning of the first four fields should be self-explanatory we will focus on the `structure` field, which holds the description of the input syntax.

The syntax description is a nested array (which in the case of TOUGH2 may contain hashes for EOS selection lists, see section 9.2.5). SITA generates input files for TOUGH2 or MARNIE by parsing through the nested syntax array recursively. The syntax parser is a subroutine that is applied to a single array of the syntax description and parses though this array in sequential order. If one of the elements is an array again, the parser will call itself recursively on this sub-array before stepping through the rest of the array. While parsing through the nested syntax array structure, SITA retrieves data from the `input` or `input-marnie` field of the test case and writes it to the input file.

KEY
output array (will generate column entries in the TOUGH2 input file)
line array (will generate a line break in the TOUGH2 input file)
key with range of validity (used to assemble references for data retrieval)
perl reference compiled at this point

```
{
    "title" : " input interface",
    "description" : "first lines",
    "version" : 1.0,
    "authors" : [ { "name" : "007", "email" : "007@grs.de" } ],
    "structure" :
    [
        "TITLE",                                              $data->{„TITLE“}
        [
          [
            ["TITLE", "A80"]                        $data->{„TITLE“}->{„TITLE“}
          ]                                                   $data->{„TITLE“}
        ]                                                               $data

        "ROCKS",                                              $data->{„ROCKS“}
        [
            [
                ["KEYWORD", "A80"]                 $data->{„ROCKS“}->{„KEYWORD“}
            ],                                                $data->{„ROCKS“}
            "MATERIAL",                       $data->{„ROCKS“}->{MATERIAL}[0]
            [
                [
                    [ "MAT", "A5" ],          $data->{„ROCKS“}->{MATERIAL}[0]->{MAT}
                    [ "NAD", "I5" ],
                    [ "DROK", "E10.4" ],
                    [ "POR", "E10.4" ],             "E10.4" is a FORTRAN printing format
                    [ "PER(1)", "E10.4" ],
                    [ "PER(2)", "E10.4" ],
                    [ "PER(3)", "E10.4" ],
                    [ "CWET", "E10.4" ],
                    [ "SPHT", "E10.4" ]
                ],
…
                [
                    [ "IRP", "I5" ],
                    [ " ", "A5" ],
                    [ "#RP(1)", "E10.4" ],                ‚#‘ marks optional parameters
…
                ],
…
            ],
            [ [ " ", "A80" ] ]                                 produces a blank line
        ],
…

        "#INDOM",                                          ‚#‘ marks optional blocks
        [ … ],
…
    ]
}
```

Fig. 9.1 Typical elements of a TOUGH2 syntax description in the input interface (JSON format)

KEY
output array (will generate column entries in the MARNIE input file)
line array (will generate a line break in the MARNIE input file)
key with range of validity (used to assemble references for data retrieval)
perl reference compiled at this point

```
 {
    "title" : " input interface",
    "description" : "first lines",
    "version" : 1.0,
    "authors" : [ { "name" : "007", "email" : "007@grs.de" } ],
    "structure" :
    [
        "GENERALDATA",                                    $data->{„GENERALDATA“}
        [
            [
                ["KEYWORD", "A80"]             $data->{„GENERALDATA“}->{„KEYWORD“}
            ],
            [
                [ "T0", "E12.5" ],
                [ "TEND", "E12.5" ],
                [ "EPS", "E12.5" ],
                [ "DT0", "E12.5" ],
                [ "DTMAX", "E12.5" ],
                [ "NXDTMAX", "I12" ]
            ],
            …
        ],
        …
        "PBLOCK",                                              $data->{„PBLOCK“}
        [
            [
                ["KEYWORD", "A80"]                  $data->{„PBLOCK“}->{„KEYWORD“}
            ],
            "MODULES",                             $data->{„PBLOCK”}->{MODULES}[0]
            [
                [
                    [ "INAMEP", "I12" ],  $data->{„ROCKS“}->{MODULES}[0]->{INAMEP}
                    [ "IPTYP", "I12" ],
                    [ "IVMLIN", "I12" ],
                    [ "IVMREC", "I12" ],
                    [ "INODE", "I12" ],
                    [ "ISUB1", "I12" ]
                ],
            …
            ],
            [ [ " ", "A80" ] ]                                  produces a blank line
        ],
…
        "#HALF-LIVES ",                                   ‚#‘ marks optional blocks
        [ … ],
…
    ]
}
```

Fig. 9.2 Typical elements of a MARNIE syntax description in the input interface (JSON format)

In order to retrieve data from the test case, SITA has to construct a Perl reference to the desired data in the test case. This reference may, for example, look like this

```
$data->{"ROCKS"}->{"MATERIAL"}[0]->{"MAT"}
```

The Perl variable `$data` holds the contents of the test case's `input` or `input-marnie` field. The given reference retrieves the name (`"MAT"`) of the first (`[0]`) material (`"MATERIAL"`) that has been declared in data block ROCKS (`"ROCKS"`).

It is the task of SITA's test case parser to construct such references. For this purpose, he collects hash keys (`ROCKS`, `MATERIAL`) and array indices (`[0]`) from the input interface while scanning the nested syntax array. The right side of Fig. 9.1 and Fig. 9.2 illustrate how the references are compiled by the parser.

9.2.1 Writing column entries with output arrays

The parser initiates a printout to the input file (which he wants to generate) if it hits a so called *output array* in the input interface. Output arrays consist of two or three scalar values (see red coloured arrays in Fig. 9.1 and Fig. 9.2). The syntax of output arrays is

```
[parameter name, printing format]
```

or

```
[parameter name, printing format, maximum number of parameter per line]
```

parameter name is the (JSON) field name under which the parameter must have been stored in the test case definition. For example, an output array with parameter `"DROK"` will require an entry like

```
"DROK" : 2600
```

in the test case unless the parameter has been marked as optional in the input interface (see below).

printing format is a string holding an (extended) Fortran format descriptor, e. g. `E10.4`, `A5`, or `I8`. If the print format is preceded by a “`+`” sign, SITA will try to maximise the precision of the number within the limits of the given format. The format descriptors `"A"` and `"Al"` will generate left-aligned output whereas `"Ar"` will create right-aligned output.

Output arrays with two arguments do not print a line break, i.e. they print a column entry to the input file.

The special output array

```
 [" " , "A10"]
```

which uses the reserved parameter name `" ",` prints a blank line (here, with 10 blanks).

Output arrays with three arguments specify a maximum number of parameters per line. For such arrays, SITA will print a list of values instead of a single value. The corresponding data entry in the test case file therefore has to be an array of numbers:

```
"X" : [2.0E+6, 1, 0, 0, 0, 21]
```

The values will be written to the input file using line breaks in order to restrict the number of values per line to the requested number.

9.2.2 Columns and lines

Line breaks are inserted at the end of arrays that contain at least one output array (directly) but aren’t output arrays themselves (i.e. they do not consist of two or three scalar values). Therefore,

```
["PAR1", "I4"]
```

creates a column entry without a line break whereas

```
[ ["PAR1","I4"] ]
```

appends a line break to the column entry. In this example, the outer brackets are responsible for the line break.

9.2.3 Grouping arrays, Keys and validity ranges

Grouping arrays are used to define validity ranges for keys. This information is needed by the parser for the construction of Perl references to test case entries.

In order to explain the use of keys, we take a look at the following abridgement of a test case:

```
    "input" : {
        "ROCKS": {
            "KEYWORD": "ROCKS”,
            "MATERIAL": [
                {
                    "MAT" : "salt",
                    "PER(1)" : 1E-20
…
                },{
                    "MAT" : "clay",
                    "PER(1)" : 1E-17
…
                }

            ]
        },
…
  }
```

In order to retrieve the parameter `"PER(1)"` for material `"salt"` from the test case, the SITA parser has to compile the expression

```
$data→{'ROCKS'}→{'MATERIAL'}[0]→{'PER(1)'} .
```

The parser collects the keys `"ROCKS"` and `"MATERIAL"` from the corresponding entries in the syntax description

```
    "structure" :
    [
…
        "ROCKS",
        [
…
            "MATERIAL",
            [
                [
                    [ "MAT", "A5" ],
                    [ "PER(1)", "E10.4" ],
…
                ],
…
            ],
…
        ],
…
```

Each key is valid only for the following array, which can be a grouping array. Grouping arrays therefore extend the validity range of a key.

There are a few restrictions regarding the use of keys. Keys must not have the names of EOS or p-modules. Further on, output arrays are not allowed to appear immediately after a key (i.e. they have to be enclosed in grouping or line breaking arrays).

For the sake of legibility, the first level keys, which correspond to data blocks in TOUGH2, should carry the name of the data blocks like, e. g. `"TITLE"`, `"ROCKS"`, or `"MULTI"`:

```
    "structure" :
    [
        "TITLE",
        [ … ],

        "ROCKS",
        [ … ],

        "MULTI",
        [ … ],
…
    ]
```

Grouping arrays do not create any printout. For this reason

```
[ ["PAR1","I4"] ]
```

generates the same output as

```
[ [ [ [ [ ["PAR1", "I4"] ] ] ] ] ]
```

9.2.4 Repeating data

While parsing the syntax description, SITA collects and discards keys as described above. In the above example, the parser has compiled the expression

```
$data→{'ROCKS'}→{'MATERIAL'}[$i]→{'PER(1) '}
```

as soon as he has reached the output array

```
 [ "PER(1)", "E10.4" ]
```

Evaluation of this expression yields the value of parameter `PER(1)`, which is stored in the test case file. The reader might have noticed that the above expression includes an array index `[$i]`, which is not part of the syntax description. This array index was inserted because the evaluation of

```
$data→{'ROCKS'}→{'MATERIAL'}
```

yielded an array (of materials) instead of a scalar value. The reason is that the test case defines an array here (which we have called a *repetition array* in chapter 8.7).

9.2.5 Optional keys and parameters

Key names and parameter names that are preceded by a number sign (#) are marked as optional. They may be omitted in the test case definition. If a key is optional and there is no corresponding entry in the test case, all entries in the validity range of that key will be ignored by the parser. Consequently, all included parameters become optional, too, even if their names are not preceded by a number sign.

In contrast, the parser is not allowed to simply skip optional parameters. Leaving out optional parameters would alter the column position of subsequent parameters. The parser therefore prints a blank string of the required length in case that an optional parameter has not been defined in the test case.

9.2.6 EOS selection lists

For TOUGH2, the user may provide a separate input interface file for each EOS module. The interface files have to be registered in the `version.json` file. However, input interfaces of different EOS modules may only differ in very few aspects. SITA therefore allows input interfaces to use EOS selection lists. EOS selection lists are JSON hashes placed immediately after key definitions:

```
          "#SELEC", {
            "EOS7":
            [ … ],
            "EOS7R":
            [ … ]
          },
```

Each time a key is followed by a hash (`{…}`), the SITA parser recognises an EOS selection list.

9.3 Output interfaces for TOUGH2

Output interfaces are only required for the TOUGH2 code and its derivatives. For the MARNIE code, simulation results are retrieved by means of the *readplot* programme (see chapter 8.6.4) which does not require an output interface.

Output interfaces are JSON files, which are stored – like the input interface – in the source code directory. Output interfaces include entries for every type of output:

```
{
  "stdout":   {…},
  "foft": {…},
  "coft": {…},
  "goft": {…},
  "doft": {…}
}
```

This is the highest level structure of the output interface. The sub-levels will be explained in the following sections.

9.3.1 Time series printouts including ELE_MAIN and CON_MAIN

Basically, interface entries for data sources `ele_main` and `con_main` correspond to those of the `foft`, `coft`, `goft`, and `doft` sources. They all follow the same syntax. Looking at a typical `foft` entry, we notice a `prefix` and an `entry` field:

```
  "foft": {
    "prefix": [
      {"par": "printout", "unit": "1"},
      {"par": "time", "unit": "a"}
    ],
    "entry": [
      {"par": "Pgas", "unit": "Pa"},
      {"par": "Pliq", "unit": "Pa"},
      {"par": "Sgas", "unit": "1"},
      {"par": "Sliq", "unit": "1"},
      {"par": "Xbrine", "unit": "1"},
      {"par": "T", "unit": "degree C"},
      {"par": "Pcap", "unit": "Pa"},
…
      {"parlist": "M_rn", "unit": "kg"},
      {"parlist": "Mmob_rn", "unit": "kg"}
    ]
  },
```

The `prefix` field informs that the first two numbers of the output refer to the printout number and the time value. The `entry` field lists all location specific parameters in sequence of their occurrence in the printout. SITA distinguishes between `par` and `parlist` fields. `par` fields define single parameters whereas `parlist` fields introduce parameter lists of arbitrary length (for this reason `parlist` fields may only occur at the end of an `entry` array).

The output interface allows the test case to access a list member by adding a number sign (#) and an index number to the root names, e.g. `M_rn#3`. If there are more than one parameter lists like in the above example, it is necessary that the parameters are listed in the following order in the time series file:

```
M_rn#1, Mmob_rn#1, M_rn#2, Mmob_rn#2, M_rn#3, Mmob_rn#3, …
```

All parameters and parameter lists include `unit` entries so that SITA can convert units if quantities with different units are plotted in the same diagram.

For the TOUGH2-GRS code, a `KDATA` value of 4 invokes an extended set of output parameters. In this case, SITA extends the name of the data source by the string `-extended` provided that this source name is defined in output interface. For example, if `foft` has been chosen as data source and `KDATA` is set to 4, SITA will search for a `foft-extended` entry in the output interface that defines an extended set of output parameters.

9.3.2 Main printout to the STDOUT channel

The interface structure for this printout reflects the way data is retrieved from the output file `OUTFILE` (the output to the `STDOUT` channel). SITA is not able to extract output parameters directly from this file due to the variable position of the printouts for the specified printout times. It would neither be time efficient nor memory efficient to parse the file sequentially for the requested output data or to read in the complete printout in advance. Instead, SITA scans the output file for time printout headers and ascertains parameter positions relative to these headers (which are always the same due to the fixed table format). By means of the output interface, SITA is told how to identify the headers and in which column each parameter can be found. It can now calculate the absolute file positions of each parameter value from the absolute header positions, the relative positions of data lines, and the column positions of parameters.

The structure of the `stdout` tells SITA how to retrieve the required information:

```
"stdout":   {

  "time": {…},

  "timestep": {…},

  "timesteplength": {…},

  "mass": {…},

  "block": [
    {…}, {…}, …
  ]
},
```

The specifications in the `time` field help SITA to find the printout headers and time values:

```
  "time": {
    "startRegEx": "OUTPUT DATA AFTER",
    "endRegEx": "ST =",
    "rowOffset": 5,
    "colOffset": 1,
    "length": 12,
    "unit": "sec"
  },
```

Here, `startRegEx` is the regular expression used to find the printout header. The beginnings of the lines matching with `startRegEx` provide the reference location for all relative file positions. The regular expression `endRegEx` makes the scan process more efficient by stopping the scan at lines matching with the value of `endRegEx`. Note that SITA uses the regular expressions of Perl. In the interface file the backslash character has to be escaped (\\) due to the JSON format (e.g. `\n` becomes `\\n`).

SITA finds the time value of a printout by means of the fields `rowOffset, colOffset`, and `length`, which hold the relative position and length of the respective string. Similar information is needed to extract the time step value:

```
  "timestep": {
    "colOffset": 14,
    "length": 6,
    "unit": "1"
  },
```

The `block` field holds one array for each output block. The number of output blocks is arbitrary but in standard TOUGH2 there are three of them: one output block for element state, a second one for advective fluxes, and a third one for diffusive fluxes. Parameter names used in the `block` field are not allowed to occur in more than one block. SITA internally enumerates output blocks according to their array position in the `block` field. This internal block number is used in error messages only.

A block is defined by all output lines printed for a specific simulation time that match with the regular expression `includeRegEx`. These lines should contain the parameter values.

```
    "block": [
      {
        "includeRegEx": "^ ....\\d ",
        "locName": { "col": 2, "length":5 },
        "locIndex": { "col": 7, "length": 6 },
        "pars": {
          "Pgas": {"col": 13, "length": 12, "unit": "Pa" },
          "Sliq": {"col": 25, "length": 12, "unit": "1" },
…
        }
      },
…
    ]
  }
```

Special care should be taken that `includeRegEx` is not too tolerant. It should only capture data lines of the current block. The `locName` and `locIndex` refer to the position of the location name and location index within a data line (locations can be elements, connections or materials). Parameter positions in a line are identified by the `col` and `length` subfields inside the `par` field. Note that all `col` fields use a column numbering starting with 1.

The `mass` field describes the structure of the mass printout:

```
    "mass": {
      "regEx": "MASS IN PLACE",
      "components": [
        {"name": "water", "rowOffset": 1, "col": 116, "length": 12,
"unit": "kg"},
        {"name": "brine", "rowOffset": 1, "col": 59, "length": 12,
"unit": "kg"},
        {"name": "main gas component", "rowOffset": 2, "col": 6,
"length": 12, "unit": "kg"}
      ]
    },
```

The regular expression `regEx` is used to find the header of the mass printout. `rowOffset` is a row offset defined relative to the header line. `col` is the column (starting with 1).

9.3.3 Time step information

The `timesteplength` field is used to extract information from the line that is printed by TOUGH2 every time step:

```
    "timesteplength": {
      "regEx": " ST = ",
      "st" : {
        "col": 25,
        "length": 12,
        "unit" : "sec"
      },
      "dt": {
        "col": 43,
        "length": 12,
        "unit": "sec"
      }
    },
```

The regular expression in field `regEx` is used to identify the data line. The field `st` holds the relative position of the time values whereas the field `dt` describes the relative location of the time step width.

10 Concluding remarks and acknowledgement

The development of SITA has been supported by the Federal Ministry for the Environment, Nature Conservation, Building and Nuclear Safety (BMUB) under the support codes UM 13 A 03400 (project ZIESEL) and 3614 R 03200 (project EMIL). The development of SITA is regarded as a long-term project. User needs and test applications will likely lead to a further development of SITA in the future.

The authors want to thank the developers of TOUGH2 hoping that SITA will contribute to the quality of this important code. Many thanks to our colleagues who dared to use the alpha versions of SITA. They gave us valuable feedback.

References

The GRS-A Reports listed below were prepared as part of research projects that were sponsored by the BMUB (formerly BMU). Quoting from these reports, reproducing them in whole or in part or making them accessible to third parties therefore requires the prior consent of the BMUB.

/ECM 11/ Ecma International: ECMAScript Languate Specification. Standard ECMA-262, Edition 5.1, 245 pp.: Geneva, 1. June 2011.

/FIS 02/ Fischer, H., Martens, K.-H.: Eingabebeschreibung des Rechenprogrammes MARNIE. GRS-A-3030, 193 pp., Gesellschaft für Anlagen- und Reaktorsicherheit (GRS) mbH: Köln, 1. June 2002.

/GHO 14/ Artifex Software Inc.: Ghostscript. Available from http://www.ghostscript.com/, cited on 10 July 2014.

/GNU 14/ GNU.org (Ed.): The GNU Make Manual, for GNU make version 4.1. 30. September 2014.

/GRS 13/ Software Management Group: Maßbahmen zur Qualitätssicherung bei der Erstellung von Computerprogrammen in der GRS (QM-richtlinien Programmentwicklung). QM-Handbuch, Teil 3: FA 03 "Fachliche Qualitätssicherung von Arbeitsergebnissen", Anlage IV. 8 pp., Gesellschaft für Anlagen- und Reaktorsicherheit (GRS) mbH: Köln, 21. November 2013.

/HOT 16/ Hotzel, S., Navarro, M., Seher, H.: QS-Handbuch für den Programmcode TOUGH2-GRS. GRS-401, ISBN 978-3-944161-82-2, Gesellschaft für Anlagen- und Reaktorsicherheit (GRS) gGmbH, 2016.

/JSON 13/ Ecma International: Einführung in JSON. Available from http://www.json.org/json-de.html, cited on 25 November 2015.

/LAT 16/ Latex Project: LaTeX documentation. as at 23 August 2015, available from https://latex-project.org/guides/, cited on 1 February 2016.

/MAR 02/ Martens, K.-H., Fischer, H., Romstedt, P.: Beschreibung des Rechenprogrammes MARNIE. GRS-A-3027, 135 pp., Gesellschaft für Anlagen- und Reaktorsicherheit (GRS) mbH: Köln, 1. January 2002.

/NAV 13/ Navarro, M.: Handbuch zum Code TOUGH2-GRS.00a. Erweiterungen des Codes TOUGH2 zur Simulation von Strömungs- und Transportprozessen in Endlagern. GRS-310, ISBN 978-3-939355-89-2, Gesellschaft für Anlagen- und Reaktorsicherheit (GRS) mbH: Köln, 2013.

/PERL 14/ Perl: The Perl Programming Language. as at 1 January 2015, available from http://www.perl.org/, cited on 10 July 2014.

/PRU 99/ Pruess, K., Oldenburg, C., Moridis, G.: TOUGH2 User's Guide, Version 2.0. LBNL-43134, 198 pp., Lawrence Berkeley National Laboratory (LBNL): Berkeley, California, USA, 1. November 1999, revised September 2012.

/WIL 16/ Williams, T., Kelley, C.: Gnuplot. 5.0.4, 2016.

/ZHA 08/ Zhang, K., Wu, Y.-S., Pruess, K.: User's Guide for TOUGH2-MP - A Massively Parallel Version of the TOUGH2 Code. LBNL-315E, Lawrence Berkeley National Laboratory (LBNL): Berkeley, California, USA, 1. May 2008.

Figures

Tables

A The JSON format

All interfaces as well as the file `version.json` use the JSON format /JSON 13/. JSON (JavaScript Object Notation) is a data format that uses human-readable text. The clarity of the format helps to reduce input data errors, which is why SITA makes extensive use of this format.

JSON is based on a subset of the JavaScript computer language with standard ECMA-262 /ECM 11/. JSON follows conventions known from the family of C based languages such as C, C++, C#, Perl, Python, etc. However, JSON is a data format and not a programming language. The supported structures are name-value pairs and ordered lists. Whitespaces may be inserted at any place between structures, strings and numbers.

Objects and values

A JSON object is an unordered hash of name-value pairs (=properties). An object starts and ends with a curly bracket. Name-value pairs are separated by commas and make use of a colon as internal separator.

```
{ name : value, name : value, … }
```

Names are strings, whereas in SITA values may be numbers, strings, arrays or objects. In this report, we use the word "field" exchangeable for a JSON name-value pair.

Arrays

An array in JSON is an ordered list of values which may be empty. It is delimited by square brackets:

```
[value, value, … ]
```

Strings

A string is a sequence of unicode characters that is delimited by quotation marks. Empty strings are permitted. Strings may include the following escape sequences:

```
\” (quotation mark)
```

```
\\ (reverse solidus)
\/ (solidus)
\b (backspace)
\f (formfeed)
\n (newline)
\r (carriage return)
\t (horizontal tab)
\uHHHH (4 hexadecimal digits)
```

Escape sequences are often needed for the regular expressions of SITA output interfaces (see section 9.3).

Numbers

JSON numbers are signed decimal numbers with an optional fractional part. It is possible to use exponential E notation. Octal or hexadecimal numbers are not permitted.

```
1E-10
1.0
-0.00001
-1E+05
```

Gesellschaft für Anlagen- und Reaktorsicherheit (GRS) gGmbH

Schwertnergasse 1
50667 Köln
Telefon +49 221 2068-0
Telefax +49 221 2068-888

Forschungszentrum
85748 Garching b. München
Telefon +49 89 32004-0
Telefax +49 89 32004-300

Kurfürstendamm 200
10719 Berlin
Telefon +49 30 88589-0
Telefax +49 30 88589-111

Theodor-Heuss-Straße 4
38122 Braunschweig
Telefon +49 531 8012-0
Telefax +49 531 8012-200

www.grs.de

ISBN 978-3-944161-81-5