User Guide

Guide for Revision R9339

Please post questions about this guide on http://starexec.lefora.com/.

Overview

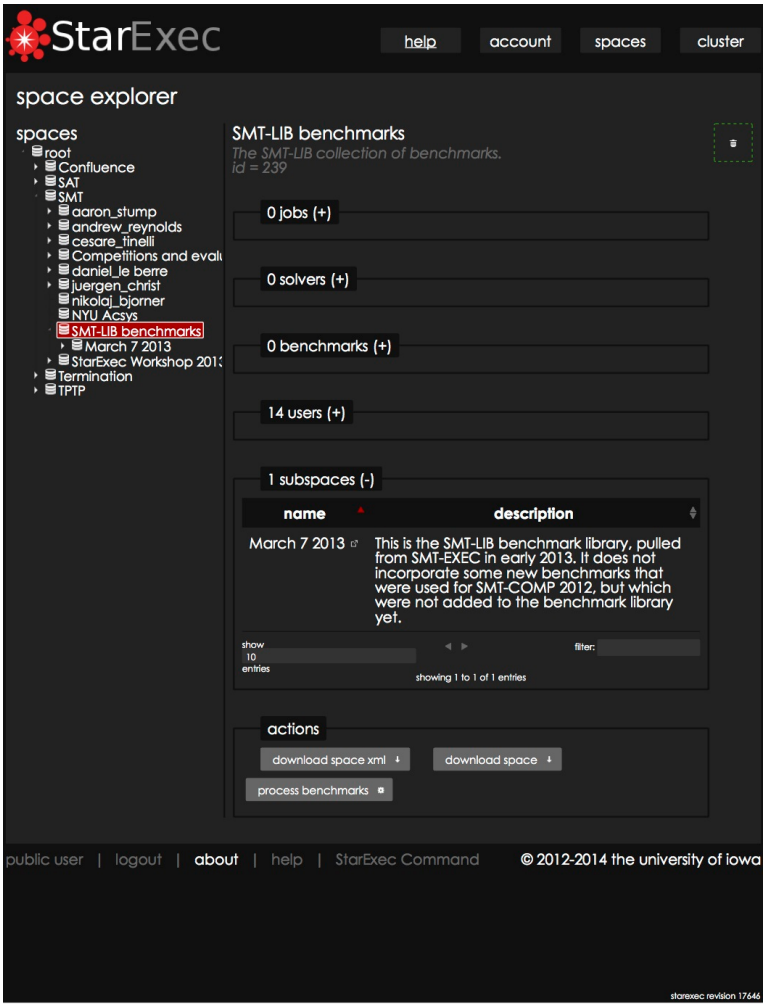
StarExec is a web-based service designed for researchers and academics to execute user-supplied programs on user-supplied input which we call *solvers* and *benchmarks* respectively. It has been designed to be utilized by a variety of logic solving communities. StarExec currently runs on a cluster of Linux Fedora nodes. This guide will help you navigate StarExec as well as accomplish the most important tasks our system supports.

Terminology

Here is a broad overview of the words and terminology we use in StarExec. It is useful to become familiar with these terms both for reading and understanding this guide and for actually utilizing StarExec.

1. **Primitives**
   Primitives are atomic entities that exist within StarExec. You can think of primitives as root classes in the StarExec taxonomy. They are mostly used to speak generally about StarExec's most basic building blocks (e.g., "a primitive's name cannot have special characters"). StarExec's primitives are *solvers*, *benchmarks*, *users*, *jobs*, *job pairs*, *spaces* and *processors*.
2. **Benchmarks**
   A benchmark is a *single text file* that is fed to a solver as input. At the time of this writing there are no plans to support multi-file benchmarks. We do however, support benchmark *dependencies,* the referencing of other benchmark files (e.g., axioms) within a benchmark.

3. **Solver**
   Solvers are programs that take in a benchmark and produce a textual output. Solvers can be uploaded as a group of files and folders, as opposed to a single executable.
4. **Configuration**
   A configuration is a single script that tells StarExec how to execute a solver. These are also known as *run scripts* or *solver configurations*. Each solver can have one or more run scripts, which allows the users to run the same solver with different flags and settings. Scripts can be written in a number of scripting languages available by default on Linux distributions (such as Bash, Perl, Python, and so on).
5. **Job**
   A job is a collection of (solver configuration, benchmark) pairs. It is basically a wrapper that contains the set of solver to benchmark mappings to be executed, or that *were* executed.
6. **Job Pair**
   A job pair consists of a solver configuration and a benchmark as mentioned above. Job pairs are atomic execution units and are associated to such information as results, CPU time, run time, etc.
7. **Processors**
   Processors are programs, uploadable only by community leaders, that take textual input and produce textual output at certain stages in the job execution pipeline. All processors to be uploaded must be contained in an file archive containing a top-level executable (a script or a binary) called `process`; top-level means it is not inside a directory other than "." in the archive. The `process` program is invoked from that same top-level directory and is free to invoke other programs contained in the archive, using a relative path within the archive. There are 3 types of processors:
   a. **Benchmark Processor**: A program invoked when uploading benchmarks (or possibly also afterwards) that takes a benchmark file name as its sole command-line argument, and produces a special attribute file which contains custom benchmark attributes.
   b. **Pre-Processor**: A program invoked when running a job, which takes a benchmark file name as its sole command-line argument and emits to `stdout` an alternate form of the benchmark to hand directly to the solver to solve.
   c. **Post-Processor**: A program that takes the name of a file containing the output of a job pair as its first command-line argument, and the path to the benchmark as its second command-line argument, and produces a special attribute file which StarExec uses to store custom job attributes.
8. **User**
   A user is a registered and verified person who belongs to at least one community and is therefore endorsed within the system by a community leader.
9. **Space**
   A space is a collection of solvers, benchmarks, jobs, users and other spaces. Spaces are similar to folders in a filesystem and are the means by which items within StarExec are organized and protected via permissions.
10. **Community**
    A community is an officially endorsed logic solving group within StarExec (e.g. Confluence, SAT, QBF, SMT, TPTP). Communities have an official *community space* to which all their members belong.
11. **Community Leader**
    A community leader is a user appointed by a StarExec administrator (or other community leaders) that has special privileges and responsibilities in managing the space of a community.
12. **Space Explorer**
    The space explorer (depicted below) is the main component of StarExec's web interface. It lets a user navigate spaces and perform most of the StarExec user functions by click, drag and drop operations, similarly to a GUI explorer for file systems.



Registration

Registration can be started here . You will receive an email from `noreply@divms.uiowa.edu` when your registration is complete in order to verify your email address. After your email address is verified, all leaders of the community which you registered for will be sent an email request. They will have the choice to either accept your request to joint their community or not. Once you are accepted in a community you will be able to log in to StarExec and access that community space. If you are rejected, you will need to re-register. You can join other communities once you are registered to one.

The registration process allows you only join existing communities. If you would like to create a new community, please contact a StarExec administrator directly.

Spaces

Navigation

You can navigate to spaces via the space explorer. At the main StarExec menu up top, hover over the Spaces menu item and select Explore from the submenu. By default you will be viewing the root space (if this is your first time visiting the space explorer).

The root space is the overall container which holds all communities and all other spaces. To see all communities you are a part of, expand all spaces under root by clicking the little tick in the lower left corner of root. This is the same way you can quickly see any subspaces of a space.

Once you have found the space you want to navigate into, simply click its name, and its details will be populated in the details panel to the right.

You can expand each item in the details panel to get more information. Click the title (e.g., 1 Jobs (+)) to show the jobs that exist in that space. Clicking the title again will hide the table below it.

In each table, there will be references to other primitives that you can click to get more information about. These will be denoted by the following "external link" icon: . Click on the item to be taken to its details page.

Permissions

Each user has a set of permissions for each space they have access to. They are as follows:

	Add	Remove
Space	Can the user create new spaces within the given space?	Can the user remove any spaces within the given space?
Job	Can the user create new jobs in the given space?	Can the user remove jobs from the given space?
User	Can the user bring new users into the given space?	Can the user remove existing non-leaders from the given space?
Benchmark	Can the user upload or copy benchmarks into the given space?	Can the user remove benchmarks from the given space?
Solver	Can the user upload or copy solvers into the given space?	Can the user remove solvers from the given space?

Space leader: In addition to the permissions defined above, a user can be a space leader. This gives him or her all the permissions above by default. Space leaders can also set the space's default permissions (see below) or remove the space. The creator of a space is that space's leader by default. Space leaders are designated by other space leaders and **can not be removed by fellow leaders, not even the original creator of the space**. A leader can change non-leaders' permissions by modifying them in the tool tip that appears when the mouse hovers over a user in the user table on the space explorer page.

Default permissions: Each space has a default set of permissions defined by the community leader which is assigned to any new user that joins the space. This permission set is defined when the space is created.

Locked space: When a space is created it can be designated as *locked* or *unlocked*. If a space is locked no one can copy items *out* of that space. This is useful if one does not want information spreading outside of one's space.

Community Leaders

A space that is a direct descendant of the root space it is considered a community space. Leaders of these spaces have special privileges. They can let new users into StarExec by approving their registration request. They can also upload new processors for their communities (although these processors are available globally across all communities). They also have all the privileges of a regular space leader on their community. Community leaders are appointed by StarExec administrators *or* other existing community leaders.

Adding New Spaces

To create a new space you must have the add space permission within an existing space. To add a subspace within a space, navigate to the space in the space explorer and click the add subspace button in the actions field set at the bottom of the details panel. Fill out the create space form and click create. You will be redirected back to the space explorer so you can see your new space.

Removing Spaces

You must have the remove space permission to remove a subspace from a space. Navigate to the space in the space explorer and click the space(s) to remove in the subspaces field set. Then drag the space to the trash can icon that appears in the upper right.

Copying Primitives

Primitives (users, solvers, benchmarks and jobs) can be copied from one space into another. This is useful if you want to collaborate or share data within StarExec. There are two prerequisites to copying items from space to space.

1. The space you are copying from, the *source space*, cannot be locked.
2. You must have the appropriate add permissions in the space you are copying to, the *target space*.

To copy an item from one space to another, navigate to the source space within the space explorer. In the right-hand panel that lists the primitives in the space (jobs, solvers, benchmarks, etc.) expand the table that contains the item(s) you want to copy. Select one or more rows in that table (the row's background color will change to indicate it is selected). Now click and drag your mouse from one of the selected rows. The space list in the left panel should expand and indicate where you can drop the items to copy. Drop the items in your target space. You will be asked to confirm, and you will get a popup if the operation was successful or not. For solvers and users, you will be given the option to copy to the entire space hierarchy rooted at your target space.

Solvers

Solvers are programs that take in a benchmark and produce a textual output. Solvers can be a group of files and folders.

Uploading

Solvers must be uploaded in one of the following archive formats: `tar`, `tar.gz`, `.tgz`, and `zip`. Your solver can contain any collection of folders and files you desire, however you must define at least one *run script* (aka configuration).

StarExec runs Red Hat Enterprise Linux (currently RHEL7), which tends to be more conservative about package versions than the Linux you might be running on your own machines. So you may find that you need to upload a statically linked executable, in order to avoid version compatibility issues.

An unbuilt solver may be uploaded and compiled by StarExec. Simply upload the solver source with a script that contains necessary steps to build the solver titled "starexec_build" in the top level directory of the compressed file. StarExec will detect this script and automatically build the solver. Please ensure that any desired configurations are still included in the /bin folder in the top level directory as you would with any precompiled solver. To see the output of the compilation, please refer to the "see build info" button, in the "actions" drop down on the solver details page for the solver in question, which

is available once the compilation has finished. Also please note that a helper configuration and benchmark are created when StarExec begins to compile the solver, however they are automatically removed once the job has finished. As with pre-compiled solvers it may be best to compile statically linked executables, and if problems persist with building the solvers it may be worthwhile to look into our provided StarExec virtual machine to investigate any problem with compilation that may be occurring.

Solver Upload Test Job

You can choose to run a "test job" when uploading a solver by selecting the "test job" option and choosing a community. A job will be created with a single job pair containing your uploaded solver and the default benchmark for the selected community.

Configurations

A configuration is a script that begins with a special prefix starexec_run_ that tells StarExec how to execute your solver. You must place your configurations in a special folder so StarExec knows where to find them. StarExec will look in a `/bin` folder in the **top level directory** of the archive you upload. The `/bin` folder will also be your working directory at the job's run time.

Anything after the underscore is treated as the configuration's name within StarExec. For example if you have two configuration files starexec_run_default and starexec_run_OtherConfig then your solver will list two configurations in StarExec called `default` and `OtherConfig`.

The run script is executed by StarExec from within the bin directory, so your script should call your solver relative to that directory. Here is an example run script below that calls Z3 which is included alongside the run script in the bin folder:

```
#!/bin/sh

./z3 -smt2 $1
```


All configurations must exist in a `bin` folder in the top level directory of the archive you upload.

Your solver should be uploaded with at least one configuration, and the system will issue a warning if no configurations are found.

It is a common mistake to zip up everything in an *wrapper* folder but that will lead to a rejected solver. For example, if you are uploading the CVC4 solver, its archive should have a `bin/starexec_run_default` file and **NOT** `cvc4/bin/starexec_run_default` where cvc4 is the wrapper folder.

Special Variables

As you can see in the script above, `$1` is a special environment variable whose value is passed to your run script when it is executed. Here is a list of arguments and environment variables your script can utilize (more will be added in the future).

Variable	Value
`$1`	The absolute path to the benchmark input file
`$2`	The absolute path to the StarExec output directory. Any files written here will be saved by StarExec so you can download them after the job is complete. Files written elsewhere are removed after each job run.
`$STAREXEC_WALLCLOCK_LIMIT`	The wall clock time limit in seconds before your job pair is terminated. This can be set at job creation time, and has a global cap at 259,200 seconds (3 days).
`$STAREXEC_CPU_LIMIT`	The CPU time limit in seconds before your job pair is terminated. This can be set at job creation time, and has a global cap at 259,200 seconds (3 days).
`$STAREXEC_MAX_MEM`	The maximum amount of memory in megabytes (without any suffix like "MB"; just a single number) which your job pair can consume before being terminated. This is user configurable and has a current limit of half the physical memory of the compute node (it is half because we hope to run two jobs per node at some point soon). For nodes in the execution queue all.q, half the memory is 128GB. For those in all2.q it is 64GB.
`$STAREXEC_MAX_WRITE`	The maximum amount of writable disk space your job pair can consume before being terminated. This is not user configurable and has a current limit of 20GB.

Benchmarks

Benchmarks are single text files that are fed to solvers when jobs are created. The name of a benchmark can be any legal file name in Linux, with a maximum length of 255 characters.

Uploading

You can upload benchmarks by navigating to your *destination* space, the space you wish to upload benchmarks to. In the action bar in the space explorer, select upload benchmarks. Fill out the benchmark upload form to complete the upload. The form contains the following fields.

- **benchmarks**:
  Select the file to upload from your local machine. Like solvers, benchmarks must be uploaded as a `.zip`, `.tar` or `.tar.gz` archive file. The archive can contain multiple files and directories. *Every* file contained in the archive will be treated as a benchmark. Please remove any readme files or metadata files before uploading your benchmarks to StarExec or they will be added to StarExec as a benchmark.
- **upload source**
  You can chose to upload either a local archive or one on the web. In the second case you must provide the archive's URL.
  upload method
  As mentioned previously, your archive may contain directories in addition to files. StarExec has two methods of dealing with directories.
  - *Convert directory structure to space structure*
    For every directory (and subdirectory) in your archive, StarExec will create a space whose name is the directory's name. It will then place all files within that directory into that space as benchmarks. Any files that are in the top level directory of the archive are placed into your chosen destination space.
  - *Place all benchmarks in space*
    This method ignores directories and recursively finds all files within the archive and adds them into the selected destination space. This option essentially *flattens* your archive.
- **default (permissions)**
  If you selected the convert file structure to space structure option, you will need to set the default permissions for all spaces that are created as a result of the benchmark upload (see *default permission* for spaces above for more details).
- **benchmark type**
  This is the type of benchmark you are uploading. The benchmark will be validated by the benchmark processor created by a community leader for that type. See *Benchmark types* below for more details.
- **downloadable**
  Select yes to allow other users who can see your benchmarks to download the benchmarks you are uploading (or otherwise see their contents). Otherwise, select no.
- **dependencies**
  Select yes if this set of benchmarks might be dependent on a previously uploaded set of benchmarks --- such as a set of axioms.
- **dependency root space**
  This is the space containing the benchmarks that yours dependent on. These benchmarks must be in the space tree rooted at this space. The paths that reference benchmarks in this tree must begin with this space or a direct child of it.
- **first directory in path corresponds to dependent bench space**
  If you have chosen a space called `Axioms` as your dependency root space and the references to this space are of the form `include('Axioms/Axiom1.txt')` or `include('Axioms/SETTHEORY/Choice.ax')`, then click yes. In contrast, if you have include statements like `include('SETTHEORY/Choice.ax')` and `include('Geometry/ParallelP.ax')` then choose no. In the latter case, `Geometry` and `SETTHEORY` must be subspaces of the dependency root space you selected.

After the archive is uploaded, you will be taken to an upload status page that will update you on the progress of processing your upload and inform you if any benchmarks fail validation.

Benchmark Types

Benchmark types are a mechanism for classifying benchmarks. A benchmark type consists of a unique name and an associated benchmark processor. This processor takes in a benchmark name as its sole command-line argument, and produces output in a StarExec defined format. The output includes an indication that the benchmark meets the requirements to be associated with the type, and it can also include any attributes about the benchmark to be stored in StarExec's database. This is useful if you wish to record additional information about benchmarks that StarExec does not directly support, such as benchmark difficulty, number of clauses, etc.

When you upload benchmarks to be classified with a given type, each benchmark is run through that type's benchmark processor.

Benchmark types are defined by community leaders and are global across all communities in StarExec. For example, if a SAT community leader creates a benchmark type, users of the SMT and TPTP community can see that type and specify their own benchmarks of that type.

Creating new benchmark types

You must be a community leader to create new benchmark types.

1. From the top main menu, go to Spaces > Communities.
2. Select the community you are a leader of and want to create a new type for.
3. In the actions panel, select edit.
4. Expand benchmark types list.
5. Select + add new in the bottom right corner of the expanded list.
6. Enter a name and a brief description for the new type and upload its corresponding processor.

To edit an existing type, click on the type's row in the expanded type list.

Creating benchmark type processors

As explained earlier, all processors are uploaded in archives which contain a top-level `process` executable. This may be a program or a script which can invoke other programs in the archive via relative path from the top-level directory of the archive. The benchmark processor is invoked with the name of the benchmark as its sole command-line argument, and it should produce output on stdout in the format described next.

Output Format

Benchmark processors must conform to a simple standard: they must take in a benchmark file and produce to `stdout` *key-value* pairs for any attributes about the benchmark that they want StarExec to save.

These attributes must be written in the Java **.properties** format where attributes are separated by newline characters and an attribute's key is separated from its value by an equal sign (=), a colon (`:`) or a space. The `.properties` format is quite general, allowing for instance keys with spaces or multi-line values. You can read more about the `.properties` format here .

Here is an example, of a possible output from a benchmark processor in the required format, with different key-value separators:

Valid Example Output

```
set-logic=LRA
status = sat
library-version 2.0
difficulty:4
```

For each line the *key* is the text on the left hand side of the separator sign (=, `:`, and/or space) and the *value* is the text on the right. *All keys and values are currently treated as strings*. There are plans in the future to allow users to query for benchmarks based on these attributes. These plans include *casting* an attribute to a numeric type to make certain queries possible such as "give me all benchmarks with difficulty greater than 3" but for the sake of producing attributes, all keys and all values are treated as strings.

Keys and values are limited to 128 characters each. The character set is ISO-8859-1, also known as Latin-1.

Validation

Benchmark processors must also produce a special attribute to let StarExec know if the processed benchmark it is a valid benchmark or not. Your processor can do any sort of validation it wants, but it must include the pre-defined `starexec-valid` key in `stdout` with a `true` or `false` value.

This attribute indicates that the benchmark passes validation:

```
starexec-valid = true
```

This attribute indicates that the benchmark fails validation:

```
starexec-valid = false
```

If the attribute is missing StarExec will treat it by default as having the value `false`.

If a benchmark is classified as valid, the benchmark will be kept and associated with the processor's type. Moreover, all its attributes except for `starexec-valid` will be stored in an internal database and associated with the benchmark. Otherwise, when a benchmark fails validation (starexec-valid = false), the entire output from the benchmark processor is saved for that benchmark. A link to that output then appears on the benchmark upload page for each benchmark that failed validation.

Another reserved key is `starexec-expected-result`. This is the value that a solver is expected to return as `starexec-result` when its job pair is processed (see post processors below).

Dependencies

If a benchmark of a given type has dependencies on other benchmarks (of the same type), its benchmark processor must inform StarExec. This is also done through a system of reserved keywords. The first is to indicate the number of dependencies.

Example:

```
starexec-dependencies = 2
```

All the dependencies must be listed and using a key of the form `starexec-dependency-`*n* where *n* is a number from 1 to the number of dependencies stated by `starexec-dependencies`. The value of the attribute is the dependency path, the local path that the solver is expecting on the execution node. This path, in Posix format, corresponds to the subspace hierarchy that contains the benchmark depended upon.

Example:

```
starexec-dependency-1 = Axioms/AGT001+0.ax

starexec-dependency-2 = Axioms/AGT001+1.ax
```

Example processor

The following script is a working benchmark processor for benchmarks in the SMT-LIB 2 format. It performs a basic validation by checking that the benchmark has a set-logic. A more realistic processor would also run syntax and type checks and perhaps also verify that it does belong to the specify logic. The script also produces a set of attributes from the benchmark, including meta-data provided in the "set-info" commands in the benchmark.

Example smt2 Benchmark Processor

```
#!/bin/bash
# AUTHOR: Tyler Jensen
# DESCRIPTION: An example benchmark processor for .smt2 benchmarks
# arg1 = the absolute path to the benchmark file

# First extract the logic family of the benchmark
LOGIC=`grep set-logic $1 | sed -e "s/(set-logic //" -e 's/[(|)]//g'`

# Here we perform basic validation: this benchmark must belong to a logic family
if [ "$LOGIC" == "" ]; then
    # The special starexec validation property
    echo "starexec-valid=false"
    exit 0
else
    # The special starexec validation property
    echo "starexec-valid=true"
fi

# Here we output the rest of the properties we wish to capture

# We want to capture the logic
echo "set-logic=$LOGIC"

# Here we process the rest of the benchmark metadata. The command below
# formats the metadata by removing invalid characters and transforming
# it into a nice key=value mapping
grep set-info $1 | sed -e "/source/d" -e "s/(set-info ://" -e "s/ /=/" -e "s/)//" -e "s/[^A-Za-z0-9_\.=-]//g"
```

The script above takes in the following benchmark (which has been truncated for the purpose of this example) and produces the output below:

Example Input

```
(set-logic QF_LRA)
(set-info :source
 |SAL benchmark suite.  Created at SRI by Bruno Dutertre, John Rushby, Maria
  Sorea, and Leonardo de Moura.

  This benchmark was automatically translated into SMT-LIB format from
  CVC format using CVC Lite.
 |
)
(set-info :smt-lib-version 2.0)
(set-info :category "industrial")
(set-info :status unsat)
(assert ...)
(check-sat)
(exit)
```

Example Output

```
starexec-valid=true
set-logic=QF_LRA
smt-lib-version=2.0
category=industrial
status=unsat
```

Uploading and Downloading XML Representations

You may find that you wish to run jobs on specific selections of benchmarks from across a wide array of spaces in your community. While you can manually drag and drop benchmarks to a new space hierarchy that you create, that process could be tedious and time-consuming. The recommended method is to create a new space hierarchy from preexisting benchmarks. You may download an XML representation of a space hierarchy that will include,

among other information, all the solver and benchmark ID's and the XML schema used to validate any uploads. If you then upload a trimmed and rearranged XML representation of a new space, StarExec will create the new space putting the preexisting benchmarks and solvers in the proper place. Note that on upload, StarExec uses the benchmark and solver IDs, but uses the space names to create new spaces. When this is done, you can now quickly run jobs on your new space hierarchy.

To see the format for the space XML files, you can consult the schema, or just download the space XML for a sample space and look through it. Note that on upload, XML attributes such as `sticky-leaders` and `inherit-users` that you set in the XML will be applied to the newly created space(s).

The format for the uploaded archive should be a .zip file containing only the .xml file (not a directory structure or other files).

Benchmark Updates in XML

You may also update benchmarks by uploading XML representations that contain Update elements inside of them. You can look at the schema to see how Updates are represented in XML. If no name is given in the XML the name will default to the original benchmark name. You can upload update processors on the edit community page. You can also get the ID of the benchmark processors and update processors needed for the XML representation by simply viewing them on this page. The update processor itself should take in two file arguments. The first file is the text needed for the update processor from the XML Text element. The second file will be the benchmark that is being updated. The update processor must produce a file named `output` which should contain the new updated benchmark. An example update processor that simply concatenates the text given in the XML to the benchmark to be updated is given below:

Update Processor

```
#!/bin/bash
cat $1 $2 > output
```

Jobs

A job is a collection of solver-configuration benchmark pairs that are grouped together as a named entity. Jobs are created within a space and will belong to that space. Jobs can be created by selecting options within the web interface, or by uploading an XML file describing the job.

Running a Job from the Web Interface

To run a job, you must have the add job permission in the space where you wish to create the job.

1. Navigate to the space where you wish to create the job.
2. In the actions bar select create job.
3. You can select a settings profile which fills in many of the following options. New settings profiles can be created by going to your own profile (under the Account menu), and selecting the edit action on the profile page.
4. Enter the job's name if desired (a default timestamped name is provided).
5. Enter a description of the job.
6. Select pre- and post processors for the job (pre-processor is likely optional, but there should be a default processor for your community).
7. Select a queue on which to run your job. all.q is the default, and unless some nodes are reserved, it will have 160 available compute nodes; all2.q has 32 nodes.
8. Enter the wall clock and cpu timeouts (for each job pair) in seconds, and the max memory in gigabytes. There are additional limits imposed on these values per queue (for the timeouts), and of course, the physical limits of the nodes. Since two jobs are run per node, one on each of the two quad-core processors, you are limited to half the physical memory. The total memory on the nodes in all2.q is 128GB, and on those in all.q is 256GB.
9. Select the desired worker queue.
   You need to select which queue the job will be submitted to. A *queue* is a collection of compute nodes (aka execution hosts). You can see the current state of the cluster by going to Cluster > Status. The default queues `all.q` and `all2.q` will submit your job to whichever node is available across the entire set of nodes of the first or second type, respectively.
10. You can indicate depth-first or round-robin creation of the job. In the first case, job pairs will be created for execution by depth-first traversal of the space hierarchy (if the later steps of creating the job will include job pairs across different spaces of the space hierarchy rooted at the current space), or by selecting a pair from each space encountered in a traversal of the hierarchy before selecting another pair from the same space.
11. You can also indicate whether you want to create the job paused (so it will not be running, and you must explicitly resume it from the page for the job to start it running), and any pre-processor seed (if your community has a pre-processor like a benchmark scrambler that wants a pseudorandom seed). You can also indicate if you want the output from your solver timestamped or not.
12. Click next.
13. Choose your desired method to run the job. A simple selection is to run and keep the hierarchy structure where each solver in every subspace is run on every benchmark in its space. If you select this, the job is immediately submitted. The following options apply only if you choose to select your solvers and benchmarks.
14. The next step is to choose your method of benchmark selection. You can run your solvers on all benchmarks in your space, all in your hierarchy, or choose a selection of benchmarks from your space.
15. Select desired solvers and configurations from the space in which your job was created.
    Select a solver by clicking its row. A selected solver will show an orange highlight. For each selected solver you should also select the configurations you want to use for that solver. You may also select all solver/configurations or all solvers with default configurations using only one click.
16. Click next or submit depending on how you decided to select benchmarks.
17. If you choose to run only select benchmarks, select the desired benchmarks you wish to include in the job by clicking on their rows.
18. Click submit. If the submission is successful, you will be redirected to the space explorer where you can view your job.

When you submit your job, it is expanded into solver-benchmark pairs which are individual execution units. These pairs are determined by pairing each selected solver with each selected benchmark. For instance, if you select 3 solvers and 4 benchmarks, your job will contain 12 job pairs.

Creating a Job by Uploading XML

There is a button in the space explorer list of actions to upload job XML. The schema for job XML is here (see also the section on solver pipelines below). Note that you must upload a compressed `.xml` file on the page for uploading job XML, not an archive containing a `.xml` file. A good way to get started with job XML is to create a small test job with the web interface, and then download the job XML for that job. There is a button to do this in the actions section of the job explorer page. Sample job XML is also provided from the page for uploading job XML.

Job Pair Execution

Solvers will receive at minimum two arguments during the execution of any job pair. First, the solver will receive the path to the job pair's benchmark. Second, the solver will receive the path to an empty directory into which the solver can save any desired output. This directory will already exist at the time of execution, and any files saved into it will be stored on Starexec and can be downloaded along with the rest of pair's output. Finally, when solver pipelines are used, a third argument will be present that contains the path to the additional output directory used by the previous stage. This directory will be empty for the first stage. Additional arguments may also be present when using solver pipelines if additional dependencies are defined.

Gathering Results

Viewing the results of a job begins in the space explorer.

1. Navigate to your job's space.
2. Expand the jobs list.
   Here you will see all jobs within the space as well as a brief overview of the state of each job.
3. Click the desired job's name to get more details.
   The job details page will show general information about the job as well as a listing of all of it pairs.
4. To get more details about a pair, click on the pair's row.
   The pair details page shows the most information about a job pair. If an attribute or statistic is unclear, hover your mouse over it to get a tool tip description.
   If the job pair was run successfully, you will have the option to view its output. Click the output list to have the job pair's combined `stdout/stderr` contents displayed in the browser. This preview shows at most 100 lines of output. To get the full output, select the popout button. You can also download individual outputs from job pairs or all of the outputs from the original job's page. Downloading output is the only way to see output that was saved in a file separate from stdout.
   You will also have the option to view the job's log within the browser. This log includes all actions that took place internally in StarExec during the various stages of the job execution pipeline. This log will be useful for debugging if you are not sure why your job failed. Select the job log list to view the pair's log.
5. You can download an entire job details page for use offline. To do so go to the job details page (see "Job details" above) and in the actions list click "download job page". After the download is completed unzip the archive and navigate to the new directory that was created. Open the job.html file with your favorite web browser.
6. If you own a job you can generate a link to that job that is accessible to those who do not have a StarExec account. On the job details page go to the "actions" section at the bottom of the page and click the "get anonymous link" button. There are three options for the type of page you can link two. The first option, "everything", allows you to anonymize the names of benchmarks, solvers, job spaces, the name of the job, etc. The second option, "everything except benchmarks", allows you to anonymize everything except for benchmark names. These two options are intended to be used for double blind studies so that the running times of solvers on benchmarks can be displayed without revealing the name of the solver. If you own the job a "solver name key" button will appear at the top of the anonymous page. This will allow you to see a mapping of solver names to their corresponding anonymized names.
   The third option "nothing" allows you to create a link that will link to your job page without any primitive names anonymized.

Adding/Deleting Job Pairs By Configuration

To add or delete job pairs by configuration, the job must be finished running or paused. To add additional job pairs or delete them go to the job's detail page accessed by clicking on the job name in the space explorer. Scroll down to "actions" and click the "add/delete job pairs" button. To delete all job pairs containing a configuration unselect the configuration and submit. When adding job pairs you have two options. You can add a new configuration that will be paired with all benchmarks that are paired with the solver's configuration in the job by selecting the "paired with solver" option. You can also pair the solver/configuration with every benchmark in the job by selecting the "all" option. If the solver is not already in the job you will only have the option to pair its configurations with all benchmarks in the job.

Pre-Processors

Pre-processors are a mechanism for modifying benchmarks right before they are submitted to solvers, during job execution. An example would be a benchmark scrambler. These processors take the name of the benchmark as the sole command-line argument, and are expected to emit the updated benchmark on `stdout`.

Pre-processors are defined by community leaders and are global across all communities in StarExec. For example, if a SAT community leader creates a pre-processor, users of SMT and TPTP can see and use the processor when creating new jobs.

Creating new pre-processors

You must be a community leader to create new pre-processors.

1. From the top main menu, go to Spaces > Communities.
2. Select the community you are a leader of and want to create a new pre-processor for.
3. In the actions panel, select edit.
4. Select + add new in the pre-processor list.

To edit an existing pre-processor, click on the processor's row in the processor list from the directions above.

Post-Processors

Post-processors are a mechanism for retrieving additional information ("attributes") from a job pair's output. The process program will be run with three command-line arguments: the name of a file containing a job pair's combined stdout/stderr, the path to the benchmark, and the path to the directory for any additional output files. They are then expected to produce textual output in a StarExec-defined format. When the job completes, StarExec scans the post-processors output and adds attributes for the job pair run to its database. This is useful if you wish to extract and collect additional information about a job pair run that is community-dependent. Any additional output files which the postprocessor write to the output files directory (third command-line argument) will also be copied back to StarExec and saved with the data from the job pair.

Post-processors are defined by community leaders and are global across all communities in StarExec. For example, if a SAT community leader creates a post-processor, users of SMT and TPTP can see and use the processor when creating new jobs.

Creating new post-processors

You must be a community leader to create new post-processors.

1. From the top main menu, go to `Spaces > Communities`.
2. Select the community you are a leader of and want to create a new pre-processor for.
3. In the actions panel, select edit .
4. Select + add new in the post-processor list.

To edit an existing post-processor, click on the processor's row in the field set from the directions above.

Creating post-processor scripts

Post-processors are uploaded in archives as described at the top of this document: there is a top-level "process" program in the "." directory of the archive, which may invoke other programs via relative path.

Output Format

Post-processors follow the same output format as benchmark processors: they must take in a solver's standard output (which is the first argument `$1` to the processor) and the path to the benchmark (this is the second argument `$2`) and produce `key=value` pairs on each line into std out.

For example, a post-processor must output all attributes about a job pair run that it wants StarExec to save in this format:

Valid Example Output

```
Active_clauses=128
Backtracking_splits=6
Binary_resolution=87
Final_active_clauses=126
Final_passive_clauses=443
```

Each attribute has a key on the left hand side of the equal side and a value on the right hand side. **All attributes are treated as strings**. There are plans in the future to be able to query for job results based on these attributes. These plans include "casting" an attribute to a numeric type to make certain queries possible such as "give me all jobs with at least 128 active clauses" but for the sake of producing attributes, all keys and all values are treated as strings.

If a pre-processor has been run on the benchmark, the processed version of the benchmark will be passed to the post-processor via the $2 argument rather than the unprocessed, original version of the benchmark.

Starexec fully supports the Java `.properties` format for processor output if you wish to produce more complicated attributes such as keys-values with spaces or multi-line values. You can read more about the `.properties` format here . Keys and values are limited to 128 characters each.

Reserved Key Word

A post processor may wish to identify one portion of the job output as the primary result. To do this, we reserve the special key word `starexec-result` s o a post processor might produce the following on a job pair's output:

```
starexec-result=unsat
```

Whatever value a job pair has for `starexec-result` appears in the results column of the job pairs table when you view the job within StarExec.

Example processor

The following script is a working TPTP post processor that is used with the Vampire solver. It extracts job attributes from Vampire's output.

Example TPTP Post-Processor

```
#!/bin/bash
# AUTHOR: Tyler Jensen
# arg1 = the absolute path to the solvers output

# Extract all lines that are key: value and format them into nice key=value pairs
grep '^[^%]\+:[A-Za-z0-9 ]\+$' $1 | sed -e 's/: /=/g' -e 's/ /_/g'
```

The script above takes in the following Vampire output (which has been truncated for the purpose of this example) and produces the output below:

Example Input

```
% remaining time: 599 next slice time: 8
dis+1_2:1_drc=off:ep=on:fde=none:gsp=input_only:lcm=predicate:nwc=1.7:ptb=off:ssec=off:sio=off:spl=backtracking:
sp=reverse_arity:updr=off_5 on ALG436-1
% SZS status Unsatisfiable for ALG436-1
% SZS output start Proof for ALG436-1
... PROOF...
------------------------------
Version: Vampire 0.6 (revision 903)
Termination reason: Refutation

Active clauses: 128
Passive clauses: 577
Generated clauses: 596
Final active clauses: 126
Final passive clauses: 443
Input clauses: 481
Initial clauses: 483

Fw subsumption resolutions: 6

Forward subsumptions: 5

Binary resolution: 87

Backtracking splits: 6
Backtracking splits refuted: 5
Backtracking splits refuted at zero level: 3

Memory used: 383KB
Time elapsed: 0.003 s
------------------------------
% Success in time 0.007 s
```

Example Output

```
Termination_reason=Refutation
Active_clauses=128
Passive_clauses=577
Generated_clauses=596
Final_active_clauses=126
Final_passive_clauses=443
Input_clauses=481
Initial_clauses=483
Fw_subsumption_resolutions=6
Forward_subsumptions=5
Binary_resolution=87
Backtracking_splits=6
Backtracking_splits_refuted=5
Backtracking_splits_refuted_at_zero_level=3
Memory_used=383KB
```

Solver Pipelines

Usually, each job pair in a job consists of a single solver configuration that executes on a single benchmark. However, it is also possible to define an ordered sequence of solver configurations that will execute one after the other in a job pair. Such a sequence is called a "solver pipeline," and each individual solver configuration that makes up the pipeline is called a "stage."

When a job pair uses a solver pipeline, there is the following control flow. The first stage in the pipeline receives the job pair's benchmark as input, exactly as any other job pair. From then on, each stage receives the stdout of the previous stage as input. Moreover, as a third argument, each stage will receive a directory containing all the previous stage's additional output (see the description of job pair execution to read a description of utilizing output files beyond stdout). In this way, solvers can be chained together. Each stage may have its own preprocessor and postprocessor, and each stage will have its output and run statistics saved individually.

When using solver pipelines, job results will be displayed by stage number. The stage number of a stage is simply the 1 indexed number indicating the order of the stage. So, the first stage has a stage number of 1, the second of 2, and so on.

Creating a Pipeline

Currently, solver pipeline functionality is supported only through creating job XML documents. Solver pipelines are defined at the same time as a job XML is uploaded to create a new job, and pairs in the new job may reference any declared pipelines.

Pipelines must be defined with a name attribute, which is referred to when creating job pairs. Pipelines consist of an ordered list of PipelineStage elements, where each PipelineStage defines a configuration to use. Exactly one PipelineStage per pipeline must be the "primary" stage; this is the stage that is used when determining the file path to place pair results at, among other things. The attribute "primary" defines the primary stage, and creating a pipeline with no primary stage or multiple primary stages will result in an error.

A pipeline may also contain noops. Noops are not executed and will not affect how the pipeline runs in any way, but they are useful for synchronizing stage numbers across different pipelines. Job pair results will be displayed by stage number in the job details page, so if you want results to be displayed together across multiple pipelines, then you may use noops to ensure that stage numbers match across different pipelines. Noops cannot be the primary stage.

Lastly, a pipeline may also define extra dependencies. Each dependency will be passed in as an extra argument in whatever order the dependencies are defined. There are two types of dependencies: dependencies on earlier stages and dependencies on other benchmarks.

Creating a dependency on an earlier stage means the output of that stage will be given. Note that because a stage will always receive the output of the stage immediately prior, stage dependencies may only be for stages at least 2 away. For example, stage 4 may have a dependency on stage 1 and 2, but not stage 3, 4, 5, and so on.

Benchmark dependencies work a bit differently. When specifying a benchmark dependency, you do not give a concrete benchmark (that is done at pair creation time). Instead, you specifying benchmark dependencies by input number. Input numbers go from 1 to *n*, where *n* is the total number of different benchmark inputs the pipeline expects. To help understand this, think of pipelines as functions that take *n* different parameters. Each stage is dependent on some of these parameters, and the values of the parameters are unique for each job pair that uses the pipeline.

An example job XML document that contains pipelines is shown below.

Example Job XML

```
<?xml version="1.0" encoding="UTF-8"?>
<tns:Jobs xmlns:tns="https://www.starexec.org/starexec/public/batchJobSchema.xsd" xmlns:xsi="http://www.w3.org
/2001/XMLSchema-instance" xsi:schemaLocation="https://www.starexec.org/starexec/public/batchJobSchema.xsd
batchJobSchema.xsd">
<SolverPipeline name="firstline">
        <PipelineStage config-id="1">
        </PipelineStage>
        <noop/>
        <PipelineStage config-id="1">
        </PipelineStage>
        <PipelineStage config-id="3" primary="true">
            <StageDependency stage="1"/>
            <BenchmarkDependency input="1"/>
            <BenchmarkDependency input="2"/>
        </PipelineStage>
</SolverPipeline>


<SolverPipeline name="secondline">
        <PipelineStage config-id="1">
        </PipelineStage>
        <noop/>
        <PipelineStage config-id="2">
        </PipelineStage>
        <PipelineStage config-id="3" primary="true">
        </PipelineStage>
        <PipelineStage config-id="1">
        </PipelineStage>
</SolverPipeline>
```

```
<Job name="another test">
<JobAttributes>
<description value="just a test"/>
<queue-id value="1"/>
<cpu-timeout value="12"/>
<wallclock-timeout value="11"/>
<mem-limit value="2.0"/>
<postproc-id value="47"/>
<preproc-id value="48"/>
</JobAttributes>

<StageAttributes>
<stage-num value="1"/>
<cpu-timeout value="2000"/>
<wallclock-timeout value="5"/>
<mem-limit value="3"/>
<stdout-save value="Save"/>
<other-save value="NoSave"/>
</StageAttributes>

<StageAttributes>
<stage-num value="3"/>
<cpu-timeout value="1"/>
<wallclock-timeout value="1000"/>
<mem-limit value="7"/>

</StageAttributes>
<JobLine pipe-name="secondline" job-space-path="space/the/cvc" bench-id="11"/>
<JobLine pipe-name="secondline" job-space-path="space/the/cvc" bench-id="15"/>
<JobLine pipe-name="firstline" job-space-path="cvc" bench-id="13">
    <BenchmarkInput bench-id="12"/>
    <BenchmarkInput bench-id="11"/>
</JobLine>
</Job>
</tns:Jobs>
```

Setting Stage Attributes

Just as a job can have global attributes like a CPU timeout, a wall clock timeout, and so on, a job may also have attributes that apply to a single stage number. Note that just as job attributes like timeouts apply across all solvers being used in the job, stage attributes apply across all pipelines. There are several attributes, but many are conceptually the same as job attributes. First, each stage number can have its own cpu timeout, wall clock timeout, and memory limit. The limits set for the overall job (in the JobAttributes tag) are treated as global limits on the execution of the pipeline, so the actual timeout enforced on a stage can depend on the runtime of previous stages. For example, say that the CPU timeout on a job is 10 seconds, and there is a pipeline with three stages, each with a CPU timeout of 5 seconds. If the first two stages of a particular job pair take 4 seconds each, then the third stage will time out after only 2 seconds, not 5.

Each stage may also have its own pre-processor and post processor, which behave exactly as they do for single-stage jobs. Note that if there is a StageAttributes tag for stage number 1, then any processors specified in the JobAttributes tag will be completely ignored. If there is no StageAttributes tag for stage 1, then the processors specified in the JobAttributes tag will be used for stage 1.

The remaining stage attributes are unique to solver pipelines. First, you can specify which components of your pair output you want to save permanently as results. This can be useful if, for example, you have a first stage which generates large inputs to your second stage where only the second stage output needs to be saved permanently. Using the 'stdout-save' tag, you may specify "Save", "NoSave", or "CreateBench." "Save" is the default, and it means your solver's stdout will be saved as usual. "NoSave" means that it will not be saved, and "CreateBench" is discussed below. You have the same set of options for any files your solver saves into the extra output directory as well.

The "CreateBench" option indicates that output should be saved both as job pair output and also as a new Starexec benchmark. In other words, the exact stdout (or extra files) that are created by your solver will become benchmarks and will be associated in a single space. If you enable saving pair output as benchmarks, you must use the "space-id" attribute. This specifies a space ID to use as root of a new space hierarchy that will contain your benchmarks. Under the space denoted by "space-id", a space hierarchy that mirrors the job space hierarchy of your job will be created, and benchmarks will be saved into this mirrored hierarchy with the same organization as the job pairs have in the job space hierarchy.

Second, there is the "bench-suffix" attribute. By default, all new job pairs created using the method above will have the same name as the input benchmark to the job pair that created the new benchmark. However, if you choose you may decide to give the new benchmarks a different suffix.

The XML document above uses all of the described attributes.

Using Pipelines in Job Pairs

Once you have defined pipelines, you can use them to create job pairs in your XML. Job pairs that do not use pipelines are created with a JobPair element, but job pairs that do use pipelines are created with a JobLine element. The JobLine element must define a pipe-name that matches a pipeline name for some pipeline created in the same document. Other than that, they are just like JobPair elements; they can specify a benchmark, a space path, and so on.

As mentioned in the section on defining pipelines, JobLine elements must specify benchmarks to use as inputs to the pipeline. A JobLine must provide exactly the same number of benchmarks that are expected by the solver pipeline being used; however, reusing the same benchmark for multiple inputs is acceptable.

Getting Results for Solver Pipelines

If a pair uses a solver pipeline, then the pair's output will be stored in a directory that contains one output file per stage, where the output file is named `<stagenumber>.txt`. So, for example, a directory might contain files like {`1.txt`, `3.txt`, `4.txt`}. Note that the numbers may have gaps due to the use of noops. On the job details page, all pair data is aggregated by stage, including the statistical views shown at the top of the page.

Installed Software and Virtual Machine

StarExec runs Red Hat Enterprise Linux (RHEL), currently Version 7. This tends to have somewhat older versions of software than distributions like Ubuntu Linux. You can explore what software is available using the virtual machine we are providing. The password for the root login on this virtual machine is `St@rexec`. In general, we cannot install additional software on StarExec unless it has a package in RHEL 7. But you can install additional software on the virtual machine, in order to compile a static binary to upload to StarExec.

Note: Python 2.7 is available on the head node and compute nodes. However, if you want to use it you have to load that version explicitly using the Linux modules system. Just add these lines to the start of a wrapper shell script before calling your solver:

Load Python 2.7

```
source /usr/share/Modules/init/sh
module load python/python27
```

GCC 7.2 can be enabled by adding the following to a solver build script:

Enable GCC 7.2

```
scl enable devtoolset-7 bash
```