The Stata Journal (2023)
23, Number 1, pp. 148–196 DOI: 10.1177/1536867X231162032

statacons: An SCons-based build tool for Stata

Raymond P. Guiteras
North Carolina State University
Raleigh, NC
rpguiter@ncsu.edu

Ahnjeong Kim
North Carolina State University
Raleigh, NC
akim7@ncsu.edu

Brian Quistorff
Bureau of Economic Analysis
Washington, DC
Brian.Quistorff@bea.gov

Clayson Shumway
North Carolina State University
Raleigh, NC
cshumwa@ncsu.edu

Abstract. In this article, we present `statacons`, an SCons-based build tool for Stata. Because of the integration of Stata and Python in recent versions of Stata, we are able to adapt SCons for Stata workflows without the use of an external shell or extensive configuration. We discuss the usefulness of build tools generally, provide examples of the use of `statacons` in Stata workflows, present key elements of the syntax of `statacons`, and discuss extensions, alternatives, and limitations. We provide recommendations for collaborative workflows and, at the end of the article, installation instructions.

Keywords: st0704, statacons, reproducible research, build tools, SCons, Python

1 Build tools

Transparency and reproducibility are increasingly important norms in empirical science (Wilson et al. 2014; Stodden, Seiler, and Ma 2018; Christensen et al. 2019; Orozco et al. 2020). Public posting of data and code, use of version control software, and literate programming are all becoming more common, both in general and with Stata users specifically.

One type of tool that has not received as much attention in Stata is build tool (Mokhov, Mitchell, and Peyton Jones 2018). A build tool tracks dependencies throughout a project: which inputs and programs are used to create which outputs, which outputs are subsequently used or combined to create final outputs, etc. A project with a workflow encoded in a build script can be replicated from beginning to end with one command. Just as importantly, when inputs or code change, a build tool can decide exactly which outputs need to be rebuilt, avoiding both errors of omission—failing to update outputs when inputs change—and errors of inclusion—unnecessarily rerunning potentially time-consuming code when no relevant inputs have changed.

Stata users already often instinctively create a rudimentary build tool: a "master" do-file that lists all a project's do-files in order. This is adequate for simple projects but has limitations in complex workflows. First, a list of do-files provides no information on what produces what, and what the relationships among different inputs and outputs

are.[1] This makes it difficult to trace outputs back to their source or debug problems. Second, a master do-file does not handle the tradeoff between errors of omission and errors of inclusion well. Running a master do-file from beginning to end should avoid errors of omission but is inefficient if some sections of the project take a long time to run. It would not make sense to rerun a computationally intensive estimation or bootstrapping procedure just to adjust the formatting of some graphs. Again, users often intuitively handle these problems by commenting out lines of the master do-file or setting up switches to skip sections of code, and this is usually fine for simple projects. Furthermore, after an initial setup, virtually all the effort in using a build tool consists of encoding inputs and outputs, so on the margin, there is little additional effort involved in using a build tool relative to providing comments in a master do-file.

However, as projects become more complex, an approach like a master do-file can be inadequate. For example, many projects have some or all of the following characteristics: merging multiple input datasets and producing several inter-related datasets for analysis; results of some analyses being used as inputs for others (for example, estimation results used in subsequent simulations); multiple collaborators working on different aspects of the project simultaneously, using version control software and file-sharing platforms to share code and files; combining Stata tasks with non-Stata tasks, for example, compiling text, tables, and figures into a PDF. In such cases, it is easy to lose track of which parts of a project need to be rebuilt. Similarly, in projects with multiple collaborators (or a single researcher collaborating with her past self), one collaborator may not know about the dependence of one part of the project on another.

Software developers face similar problems and have developed sophisticated build tools to deal with them. Two ideas are central: first, the relationships between inputs, code, and outputs are encoded in a build script; second, the build software uses this script and information it collects on the state of the project to manage the build, deciding what is up to date and what needs to be built or rebuilt. The build script itself provides living documentation of the full structure of the project, and the build tool can provide the user with the status of any particular component, or of all components.

The use of build tools does not appear to be common among Stata users. We suspect the main barriers to adoption are, first, lack of awareness of the existence and value of build tools; second, the start-up cost of configuring the tool, system, and Stata so that they can work together; and third, the start-up cost of learning how to use these tools.

In this article and with `statacons`, we attempt to address all three of these barriers. First, we will show in our examples that `statacons` can be used to manage both simple and complex workflows efficiently. Second, because of the integration of Python and Stata in recent versions (16+), as well as our own adaptations, we are able to make SCons, a popular Python-based build tool, accessible in Stata through our command `statacons`. With a minimal amount of configuration, Stata users can use `statacons` directly in Stata and write the build scripts, `SConstruct`s, directly in Stata's Do-file

1. Comments in a master do-file can provide some documentation of inputs and outputs, but drift between comments and the true content of code is a notorious source of confusion and error (Gentzkow and Shapiro 2014).

Editor, taking advantage of the Editor's Python syntax highlighting.[2] Third, our examples in this article, as well as additional examples in our companion web tutorial and project Wiki, can easily be adapted to users' own work.

More specifically, `statacons` offers the following benefits: a) not requiring changes to existing Stata code or the structure of existing workflows; b) rebuilding exactly those parts of a project that need to be rebuilt at any given moment, with neither errors of inclusion nor omission; c) complementing existing workflow practices, including literate programming and use of version control software and file-sharing platforms; d) allowing easy extensibility; and e) operating fully within Stata.

Our article proceeds as follows: In section 2, we use a simple example of managing a Stata workflow with `statacons` and an `SConstruct` to introduce basic concepts, vocabulary, and mechanics. In section 3, we provide the syntax of the `statacons` command and `SConstruct`s. In section 4, we present an applied example of using `statacons` to manage the complex workflow of an empirical project (Shumway and Wilson 2022). In section 5, we describe some of the technical details of our adaptation of SCons to Stata. In section 6, we discuss extensions, alternative approaches, and limitations. Section 7 provides recommendations for collaborative workflows. Section 8 concludes.

We include a detailed installation guide at the end of the article. Online appendices discuss some additional advanced features, such as parallel builds. Our project webpage[3] provides installation instructions and hosts a companion web tutorial based on Software Carpentry's "Automation and Make" tutorial (Jackson 2016) with several additional worked examples.[4] The `statacons` package is available at our project repository,[5] where we have posted complete replication code and data for the *Introductory example* and key code from the *Applied example*, as well as a project Wiki with examples of additional tools.

2 Introduction by example

In this section, we use a simple example to introduce the basic concepts, terminology, and mechanics of managing a workflow with `statacons`. Replication code and data for this example are provided in `stataconsIntro.zip`, posted to our repository.[6]

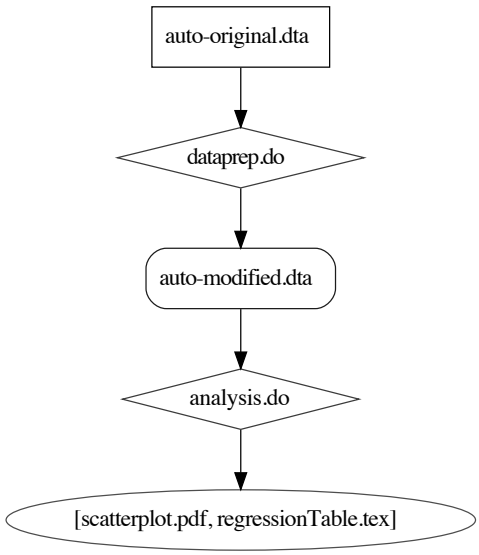
Listing 1 shows the directory structure and key files. Our project has two steps, using the two do-files in listing 1. In the first step, the do-file `dataprep.do` takes an input file, `auto-original.dta`; creates a new variable, `mpg_sqd`, which is the square of `mpg`; and saves the resulting dataset as `auto-modified.dta`. In the second step, the do-file `analysis.do` uses `auto-modified.dta` to create a scatterplot of `price` against

2. `SConstruct` files written for `statacons` are standard `SConstruct`s and will work without modification in standard SCons, so users who prefer the terminal can continue to work there.
3. https://bquistorff.github.io/statacons
4. https://bquistorff.github.io/statacons/swc
5. https://github.com/bquistorff/statacons
6. https://github.com/bquistorff/statacons/raw/main/examples/stataconsIntro.zip

mpg, saved as scatterplot.pdf, and regresses price on mpg and mpg_sqd, exporting the regression results to regressionTable.tex using eststo and esttab (Jann 2007, 2014). This workflow is represented visually in figure 1 and can easily be managed by the master do-file master.do at the bottom of that figure.

```
introExample/
   master.do
   SConstruct
   code/
      dataprep.do
      analysis.do
   inputs/
      auto-original.dta
   outputs/
      auto-modified.dta
      scatterplot.pdf
      regressionTable.tex

// dataprep.do
version 17

use inputs/auto-original

generate mpg_sqd = mpg^2
label variable mpg_sqd "Mileage (mpg) squared"

save outputs/auto-modified, replace

// analysis.do
version 17

use outputs/auto-modified

twoway scatter price mpg
graph export "outputs/scatterplot.pdf", replace

regress price mpg
eststo linear

regress price mpg mpg_sqd
eststo quadratic

esttab linear quadratic using "outputs/regressionTable.tex", ///
  se r2 replace
```

Listing 1. File structure and do-files for introductory example




```
// master.do
version 17

do "code/dataprep.do"
do "code/analysis.do"

exit
```

Figure 1. Linear workflow and master do-file for introductory example

When using a build tool, it is useful to think about the workflow differently. Rather than thinking about a sequence of do-files in a particular order, we think about the targets we want to build and the dependencies we use to create these targets. In this example, the ultimate targets are `scatterplot.pdf` and `regressionTable.tex`. The dependencies for these targets are the source code `analysis.do` and `auto-modified.dta`, which is itself a target with dependencies `dataprep.do` and `auto-original.dta`.

This workflow is depicted visually in figure 2. To use `statacons`, we encode these relationships in an `SConstruct` file. The `SConstruct` file used in this example is shown in listing 2.

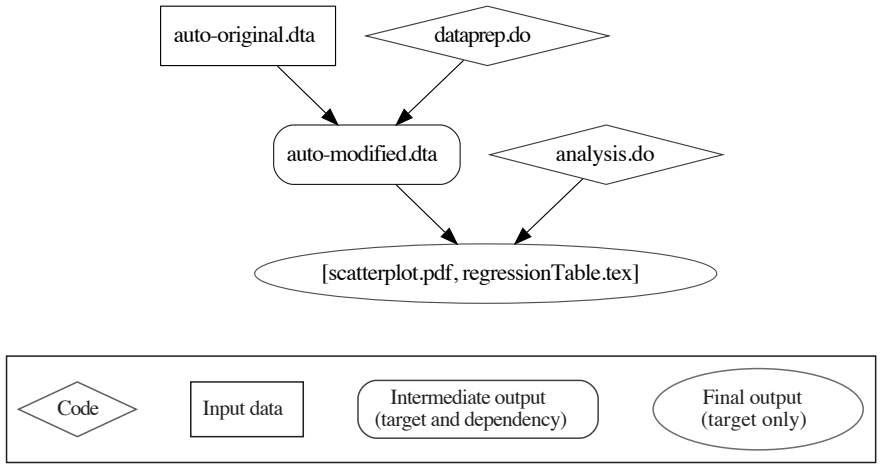


Figure 2. Target-build workflow for introductory example

```
# **** Setup from pystatacons package *****
import pystatacons
env = pystatacons.init_env()
# **** Substance begins        *****
# analysis
cmd_analysis = env.StataBuild(
    target = ['outputs/scatterplot.pdf',
              'outputs/regressiontable.tex'],
    source = 'code/analysis.do'
)
Depends(cmd_analysis, ['outputs/auto-modified.dta'])
# dataprep
cmd_dataprep = env.StataBuild(
    target = ['outputs/auto-modified.dta'],
    source = 'code/dataprep.do'
)
Depends(cmd_dataprep, ['inputs/auto-original.dta'])
```

Listing 2. `SConstruct` file for introductory example

Let us first consider the analysis task, which we have named `cmd_analysis`.[7] The targets are `scatterplot.pdf` and `regressionTable.tex`. The source file for this

7. More precisely, `cmd_analysis` is a *NodeList*, where the nodes are the targets. Conveniently, you can use the *NodeList* later in the `SConstruct` to refer to the targets. That is, if we use `cmd_analysis` elsewhere in this `SConstruct`, SCons will understand this to mean the targets `scatterplot.pdf` and `regressionTable.tex`.

task is `analysis.do`. The source file is the code we will run to build these targets. The `Depends()` line lists any dependencies other than the source file,[8] which in this case is `auto-modified.dta`. Finally, `env.StataBuild` tells SCons that this is a task for Stata, to be handled in the way defined by the builder method `StataBuild`. The essence of `StataBuild` is that SCons will build the targets (`scatterplot.pdf` and `regressionTable.tex`) by running the source file (`analysis.do`) in Stata's batch mode.[9]

The data prep task, `cmd_dataprep`, is similar, and we leave it as an exercise.

We now use `statacons` to build our outputs. We run `statacons` with the option `debug(explain)`, which adds some helpful screen output:

```
. statacons, debug(explain)
scons: Reading SConscript files ...
Using 'LabelsFormatsOnly' custom_datasignature.
Calculates timestamp-independent checksum of dataset,
  including variable formats, variable labels and value labels.
Edit use_custom_datasignature in config_project.ini to change.
  (other options are Strict, DataOnly, False)
scons: done reading SConscript files.
scons: Building targets ...
scons: building `outputs\auto-modified.dta' because it doesn't exist
stata_run(["outputs\auto-modified.dta"], ["code\dataprep.do"])
Running: "C:\Program Files\Stata17\StataMP-64.exe" /e do "code\dataprep.do".
  Starting in hidden desktop (pid=37160).
scons: building `outputs\scatterplot.pdf' because it doesn't exist
stata_run(["outputs\scatterplot.pdf", "outputs\regressionTable.tex"],
> ["code\analysis.do"])
Running: "C:\Program Files\Stata17\StataMP-64.exe" /e do "code\analysis.do".
  Starting in hidden desktop (pid=31392).
scons: done building targets.
```

We now highlight a few key lines from the screen output.

```
scons: building `outputs\auto-modified.dta' because it doesn't exist
```

This line tells us SCons has noticed that `auto-modified.dta` does not exist and must be created. The way it is created is given in the next two lines:

```
stata_run(["outputs\auto-modified.dta"], ["code\dataprep.do"])
Running: "C:\Program Files\Stata17\StataMP-64.exe" /e do "code\dataprep.do"
```

8. While not strictly necessary, it is useful to split the source from the other dependencies so that there is no ambiguity about what code will be used in the `StataBuild` environment.
9. Batch mode does not require user input and is often used for automating Stata tasks. See the "Stata batch mode" section in appendix B of the relevant *Getting Started with Stata* manual.

The first of these is sent to the Stata builder in SCons: in words, "build the target `auto-modified.dta` using the source `dataprep.do` in the way defined in the Stata builder." The second line is what the builder tells the computer to do: run the do-file `dataprep.do` in batch mode.[10]

After creating `auto-modified.dta`, SCons sees that `scatterplot.pdf`, its next target, does not exist and needs to be built, which it does by running `analysis.do` in batch mode. SCons then determines that all targets are up to date and so it exits.

Incidentally, the order in which the tasks appear in the `SConstruct` is not important. Even though the `SConstruct` lists the analysis task first, the logic of the build requires that SCons perform the data prep task first.[11]

Now if we ask `statacons` to rebuild, it will check whether the dependencies of any target have changed since the previous build.

```
. statacons, debug(explain)
scons: Reading SConscript files ...
Using 'LabelsFormatsOnly' custom_datasignature.
Calculates timestamp-independent checksum of dataset,
  including variable formats, variable labels and value labels.
Edit use_custom_datasignature in config_project.ini to change.
  (other options are Strict, DataOnly, False)
scons: done reading SConscript files.
scons: Building targets ...
scons: `.' is up to date.
scons: done building targets.
```

`statacons` does this by calculating file signatures—a compact string that will not change if the file contents do not change, and likely will change if the file contents do—for each target and dependency and comparing these signatures to those from the previous build, which it has saved in a database called `.sconsign.dblite`.[12] The `stataconsign` command (a Stata call to the SCons command `sconsign`) will print the content of the `.sconsign.dblite` database.

10. Exactly where the Stata executable is on your machine will differ from user to user. We have written `statacons` to find it automatically for most standard setups and to allow users to adjust the default in configuration files. We discuss configuration files, by default named `config_project.ini` and `config_local.ini`, at greater length in section 3.4.
11. Of course, the way the `SConstruct` is written affects how easy it is for a human reader to understand. In some cases, it may be easier to follow if the main outputs are listed first, as we have done here. In other cases, a chronological ordering may be preferable.
12. Technically, there are different types of functions: hashes, which provide some cryptographic guarantees, and checksums, which do not. In the text, we use the more generic term "signature" for all of these functions.

As an example, here is the current state of the `.sconsign.dblite` database for this *Introductory example*:

```
. stataconsign
=== .:
SConstruct: None 1651866262 581
=== code:
analysis.do: af3fe510ee8586a10401fe8e9b4fef49 1652884882 317
dataprep.do: 57d511b57ba3f9d41a1c488dab10b3ff 1652884882 192
=== inputs:
auto-original.dta: 74:12(71728):3831085005:1395876116:17340132 1640899314 12765
=== outputs:
auto-modified.dta: 74:13(15616):2430311699:721316426:3189221131 1652884885 13701
        code/dataprep.do: 57d511b57ba3f9d41a1c488dab10b3ff 1652884882 192
        inputs/auto-original.dta: 74:12(71728):3831085005:1395876116:17340132
> 1640899314 12765
        947c32d25892d9cb39480bf147f78ed3 [stata_run(target, source, env)]
regressionTable.tex: bb4d5242836537f8cb562435b9e17cb8 1652884889 931
        code/analysis.do: af3fe510ee8586a10401fe8e9b4fef49 1652884882 317
        outputs/auto-modified.dta: 74:13(15616):2430311699:721316426:3189221131
> 1652884885 13701
        947c32d25892d9cb39480bf147f78ed3 [stata_run(target, source, env)]
scatterplot.pdf: 5e43c38cd77e29c7235c2866a448742f 1652884889 65452
        code/analysis.do: af3fe510ee8586a10401fe8e9b4fef49 1652884882 317
        outputs/auto-modified.dta: 74:13(15616):2430311699:721316426:3189221131
> 1652884885 13701
        947c32d25892d9cb39480bf147f78ed3 [stata_run(target, source, env)]
```

Examining the database provides insight into how SCons and `statacons` work. For the code (`analysis.do` and `dataprep.do`), the database is straightforward. Each is listed with three elements: a signature, a timestamp, and the file length. Similarly, for the input dataset `auto-original.dta`, the dataset entry consists of the same three items, although the signature is calculated differently, as we discuss in section 5.

For generated files, that is, files that are built by `statacons`, the database entries are more complex. As an example, consider the line for `auto-modified.dta`. Recall that this is the output of the data prep task. The first line in the database entry for `auto-modified.dta` has the same structure as those of the code and input data: a signature, a timestamp, and the file length. The entry for `auto-modified.dta` has three additional lines: one for each of the two dependencies (`auto-original.dta` and `dataprep.do`) and one for the build action `stata_run`. The lines for the dependencies just replicate their own database entries. The final line is a signature for the build action. These are the three items that SCons needs to consider when deciding whether to rebuild `auto-modified.dta`. If any of the dependencies or if the build action have changed, SCons will decide to update `auto-modified.dta`. If none have changed, SCons will know that it does not need to rebuild. The remaining entries are all for derived files and all have the same structure: one line for the file itself, one line for each dependency, and one line for the build action.

In this case, `statacons` determines that no dependencies have changed since the last build, so nothing needs to be done.

Now let us see what `statacons` does when we make a small modification only to `analysis.do`. We add a title to the scatterplot, changing

```
twoway scatter price mpg
```

to

```
twoway scatter price mpg, title("Price versus MPG")
```

What happens when we run `statacons`?

```
. statacons, debug(explain)
scons: Reading SConscript files ...
Using 'LabelsFormatsOnly' custom_datasignature.
Calculates timestamp-independent checksum of dataset,
  including variable formats, variable labels and value labels.
Edit use_custom_datasignature in config_project.ini to change.
  (other options are Strict, DataOnly, False)
scons: done reading SConscript files.
scons: Building targets ...
scons: rebuilding `outputs\scatterplot.pdf' because `code\analysis.do' changed
stata_run(["outputs\scatterplot.pdf", "outputs\regressionTable.tex"],
> ["code\analysis.do"])
Running: "C:\Program Files\Stata17\StataMP-64.exe" /e do "code\analysis.do".
  Starting in hidden desktop (pid=196).
scons: done building targets.
```

`statacons` sees that we need to rebuild our target `scatterplot.pdf`, because its source file `analysis.do` has changed. `statacons` also sees that there have been no changes to the dependencies for `auto-modified.dta`, so it does not need to rebuild that.

Let us next explore what happens when we edit `dataprep.do`. We add a variable, `mpg_cub`, the cube of `mpg`:

```
generate mpg_cub = mpg^3
label variable mpg_cub "Mileage (mpg) cubed"
```

And we rebuild using `statacons`:

```
. statacons, debug(explain)
scons: Reading SConscript files ...
Using 'LabelsFormatsOnly' custom_datasignature.
Calculates timestamp-independent checksum of dataset,
  including variable formats, variable labels and value labels.
Edit use_custom_datasignature in config_project.ini to change.
  (other options are Strict, DataOnly, False)
scons: done reading SConscript files.
scons: Building targets ...
scons: rebuilding `outputs\auto-modified.dta' because `code\dataprep.do' changed
stata_run(["outputs\auto-modified.dta"], ["code\dataprep.do"])
Running: "C:\Program Files\Stata17\StataMP-64.exe" /e do "code\dataprep.do".
  Starting in hidden desktop (pid=28972).
scons: rebuilding `outputs\scatterplot.pdf' because `outputs\auto-modified.dta'
> changed
stata_run(["outputs\scatterplot.pdf", "outputs\regressionTable.tex"],
> ["code\analysis.do"])
Running: "C:\Program Files\Stata17\StataMP-64.exe" /e do "code\analysis.do".
  Starting in hidden desktop (pid=18744).
scons: done building targets.
```

First, `statacons` sees that `dataprep.do` changed, so its source, `auto_modified.dta`, must be rebuilt. Then, after rebuilding `auto_modified.dta`, `statacons` sees that it changed, and because `auto_modified.dta` is a dependency for our ultimate targets `scatterplot.pdf` and `regressionTable.tex`, these targets must be rebuilt.[13]

Finally, let's make a small edit to `dataprep.do` that does not change the content of `auto-modified.dta`:

```
// this is an edit to dataprep.do that does not change auto-modified.dta
```

The result of rebuilding with `statacons` may be surprising:

```
. statacons , debug(explain)
scons: Reading SConscript files ...
Using 'LabelsFormatsOnly' custom_datasignature.
Calculates timestamp-independent checksum of dataset,
  including variable formats, variable labels and value labels.
Edit use_custom_datasignature in config_project.ini to change.
  (other options are Strict, DataOnly, False)
scons: done reading SConscript files.
scons: Building targets ...
scons: rebuilding `outputs\auto-modified.dta' because `code\dataprep.do' changed
stata_run(["outputs\auto-modified.dta"], ["code\dataprep.do"])
Running: "C:\Program Files\Stata17\StataMP-64.exe" /e do "code\dataprep.do".
  Starting in hidden desktop (pid=42216).
scons: done building targets.
```

13. Careful readers of the underlying do-files might note that `analysis.do` does not actually use the variable `mpg_cub`. However, `statacons` does not know that—it only sees that a dependency has changed and, because the targets potentially could change, they must be rebuilt. See section 4 for a strategy to avoid this sort of quasi-false positive.

This highlights an important way in which SCons, and therefore `statacons`, differs from the classic and widely used build tool `make` and many of its derivatives.[14] As we would expect, `statacons` sees that `dataprep.do` has changed, so `auto-modified.dta` must be rebuilt. However, notice that `statacons` does not rebuild `scatterplot.pdf` and `regressionTable.tex`. This is because, while `auto-modified.dta` has been rebuilt, nothing about it has changed other than its timestamp. `statacons` will rebuild a target only if a dependency has changed. When deciding whether to rebuild `scatterplot.pdf` and `regressionTable.tex`, `statacons` computes a signature of `auto-modified.dta` that depends only on the content of the dataset, not on its timestamp. Then `statacons` will check whether this signature has changed since the previous time the target was built. Because none of the content of `auto-modified.dta` has changed, its signature is unchanged, and `statacons` sees that there is no need to rebuild any more targets.

3 Syntax

The syntax of `statacons` follows that of SCons. We also allow users to use the standard SCons syntax, although not a mix of the two in the same command.[15] For the most commonly used options, we have adapted SCons syntax into a style more familiar to Stata users. These options are listed below.

The full syntax of SCons is too long to replicate here but can be found in the SCons manual (SCons Development Team 2021a). Here we present the basic syntax and several of the more useful options, which we have made available in Stata syntax. For options not listed here, the user will need to use standard SCons syntax. More in-depth and comprehensive discussion can be found in the *SCons User Guide* (SCons Development Team 2021b).

`statacons` works in Stata 16 and later.

The syntax of `statacons` is

`statacons` [*targets*] [`,` *options*]

The SCons equivalent is

`statacons` [*options*[`=` *values*]] [*targets*]

By default, `statacons` will look in the current directory for an `SConstruct` file, consider all targets described by that file, and then build or rebuild targets when needed.

14. For example, GNU Make: https://www.gnu.org/software/make/.
15. Invoking SCons from a terminal prompt requires SCons syntax.

3.1 Selecting targets

The user can select a single target or a subset of targets by specifying them on the command line. In the example from section 2, to rebuild only `auto-modified.dta` but not the other targets, we would use

```
. statacons outputs/auto-modified.dta
```

The SCons equivalent is identical.

Alternatively, the user can edit the `SConstruct` file to specify the default targets with the `Default()` function. See section 3.3, *SConstruct syntax*, below.

If no targets are specified on the command line and no `Default()` function is given in `SConstruct`, then `statacons` will consider all targets in `SConstruct` in the current directory or subdirectories.

3.2 Options

3.2.1 Standard SCons options

We discuss several of the most commonly used options. For a complete list of options, see the SCons manual (SCons Development Team 2021a).

`clean` will clean—that is, delete—specified targets. Be careful when using this option, because by default it will remove all targets defined in `SConstruct`.

SCons equivalents: `-c`, `--clean`, `--remove`

`debug(`*type*[`,` *type*`...`]`)` will print additional information to the screen to help debug the build process. What information is printed depends on the *type* or *type*s specified. The most useful *type* is `explain`, which will request that `statacons` print an explanation for each target it selects to build or rebuild—typically, which dependency has changed. (Explanations are not given for targets that are not built.)

SCons equivalent: `--debug=`*type*[`,` *type*`...`]

`directory(`*directory*`)` will change the directory to *directory* before starting to build. The change of directory will occur only within the SCons process; the user's working directory will not change in Stata itself.

SCons equivalents: `-C` *directory*, `--directory=`*directory*

`dry_run` will put SCons into "no execute" mode. SCons will attempt to determine what targets it would build and print the commands it would execute, but it will not actually execute these commands.

Because SCons by default rebuilds targets only when dependencies have changed, SCons may not be able to determine whether it will need to rebuild "downstream" targets. For example, suppose `A.dta` is a dependency of `B.dta` and `B.dta` is a dependency of `C.dta`. When `A.dta` changes, SCons knows it must rebuild `B.dta`.

But in "no execute" mode, SCons does not actually rebuild `B.dta`, so it does not know whether `B.dta` will change. Thus, in "no execute" mode, SCons does not know whether it will have to rebuild `C.dta` and will not print any commands for rebuilding `C.dta`. In general, we can think of the default set of commands printed in "no execute" mode as the smallest set that will be executed.

SCons equivalents: `-n`, `--no-exec`, `--dry-run`, `--just-print`

`file(`*file*`)` or `sconstruct(`*file*`)` allows the user to specify *file* as the initial `SConstruct` file for `statacons` to read. By default, `statacons` looks for a file named `SConstruct` in the current directory.

SCons equivalents: `-f` *file*, `--file=`*file*, `--sconstruct=`*file*

`help` prints the full SCons help message to the screen. The user can define an additional help message in the `SConstruct` by using the `Help()` function; see the SCons manual (SCons Development Team 2021a) for details. (`statacons, help` is distinct from `help statacons`; the latter gives the help file for the Stata command, not the SCons program.)

SCons equivalents: `-h`, `--help`

`q` (for "quiet") requests no printed SCons status messages.

SCons equivalent: `-Q`

`silent` requests no printed messages about SCons actions nor SCons status messages. `silent` implies `q`.

SCons equivalents: `-s`, `--silent`, `--quiet`

`tree(`*type*[`,` *type*...]`)` prints a dependency tree, which is useful for debugging or checking on the status of a build, or just for visualizing and understanding the workflow. The part of the tree printed and the format depend on the *type* or *type*s specified:

`all` prints the entire tree.

`derived` prints only "derived" (that is, target) files. It will omit files that are not targets, for example, code or input data.

`linedraw` uses Unicode characters to draw the tree rather than the default ASCII. Some users may find this easier to read on the screen.

`status` prints the current status of each item on the tree. This is useful for understanding the status of a build, because it will tell you which files are current (`C`), which files exist (`E`), and several other status categories.

`prune` makes the tree easier to read by not repeating dependencies for files that have already been described by the tree. Without this option, `tree()` prints a lot of redundant information. With the `prune` option invoked, `tree()` will indicate that a file's dependencies have already been listed by enclosing the filename in square brackets.

SCons equivalent: `--tree=`*type*[, *type*...]

We most commonly invoke `tree()` as `tree(status, prune)`. For our introductory example, this produces the output shown in listing 3.

Exporting the output of `tree()` to a text file[16] will produce output similar to that of the BITSS / Social Science Reproduction Platform Diagram Builder (BITSS 2020).

```
. statacons, tree(status, prune) dry_run silent
 E         = exists
  R        = exists in repository only
   b       = implicit builder
   B       = explicit builder
    S      = side effect
     P     = precious
      A    = always build
       C   = current
        N  = no clean
         H = no cache
[E b   C  ]+-.
[E b   C  ]  +-code
[E     C  ]  | +-code\analysis.do
[E     C  ]  | +-code\dataprep.do
[E b   C  ]  +-inputs
[E     C  ]  | +-inputs\auto-original.dta
[E b   C  ]  +-outputs
[E B P C  ]  | +-outputs\auto-modified.dta
[E     C  ]  | | +-code\dataprep.do
[E     C  ]  | | +-inputs\auto-original.dta
[E B P C  ]  | +-outputs\regressionTable.tex
[E     C  ]  | | +-code\analysis.do
[E B P C  ]  | | +-[outputs\auto-modified.dta]
[E B P C  ]  | +-outputs\scatterplot.pdf
[E     C  ]  |   +-code\analysis.do
[E B P C  ]  |   +-[outputs\auto-modified.dta]
[E     C  ]  +-SConstruct
```

Listing 3. SCons tree from our introductory example

16. For example, in a Unix-like terminal, type `scons -ns --tree=prune > treeOutput.txt`.

3.2.2 Custom statacons options

All the options above are standard to SCons. We have written several options specifically for `statacons`.[17]

`assume_built("`*target*`")` will instruct the Stata builder to skip a task if all of its targets are listed and then to mark those targets as up-to-date. This can be a colon-separated list of file patterns.

SCons syntax: `--assume-built="`*target*`"`

`assume_done("`*filename*`.do")` will instruct the Stata builder to skip the given do-file in the current build but mark the associated target(s) as up-to-date. This is useful if we know that a target is up-to-date but SCons does not recognize it as such, for example, 1) if we have just built `target.tex` by running `filename.do` directly in Stata rather than through SCons or `statacons`; and 2) if we have only made edits that we know will not result in any changes to `target.tex`, such as adding comments in `filename.do`. This works even if the task for this do-file includes parameters or a non-do-file command. See the discussion in section 6.3, *Limitations*.

SCons syntax: `--assume-done="`*filename*`.do"`

`assume_done("`*filename1*`.do:`*filename2*`.do[ : ... ]")` will split by the colon and skip each file. Each component can be a file pattern, as seen in the next example.

SCons syntax: `--assume-done="`*filename1*`.do:`*filename2*`.do[ : ... ]"`

`assume_done(*)` will skip all do-files in the current build and mark all of their targets as up-to-date.

SCons syntax: `--assume-done=*`

`config_file(`*config_file*`)`, by default, `statacons` will look for `config_project.ini` and `config_local.ini` in the same directory as the `SConstruct`. If these files do not exist or they do not specify several core parameters, such as the location of the Stata executable, then defaults based on typical configurations for the user's operating system, version, and edition of Stata will be used. Setting `config_file(`*config_file*`)` will instruct the Stata builder to use *config_file* as the configuration file. See section 3.4 for more on default settings and configuration files. The user can specify multiple files as *file1*`:`*file2*`:...:`*filen*. In case of conflicts between files, the file that appears later in *file1*`:`*file2*`:...:`*filen* will take precedence.

SCons syntax: `--config-file=`*config_file*

17. Standard SCons (that is, invoked as SCons from the command line) will process these commands as long as you have installed the Python package `pystatacons` and your `SConstruct` imports `pystatacons`. SCons-style syntax will be required.

`show_config` will print the current values of all configuration options, whether chosen by default or specified in a configuration file. `statacons, show_config` activates the `dry_run` and `silent` options, so `statacons` will not build any files or report anything about the state of the build.

SCons syntax: `--show-config`

3.3 SConstruct syntax

3.3.1 Basic SConstruct recipe

The basic recipe for encoding a target built by Stata into an `SConstruct` file is

```
task_name = env.StataBuild(
      target = ['path/to/target1.ext', 'path/to/target2.ext'],
      source = 'path/to/dofile.do'
)
Depends(task_name, ['path/to/dependency1.ext',
                    'path/to/dependency2.ext']
)
```

In section 2, we described each of the elements: builder (here `StataBuild`), target, source, and dependencies. The target and the source are both required.[18]

3.3.2 Additional options for SConstruct recipe

We have added some options to the basic recipe:

```
task_name = env.StataBuild(
      target = ['path/to/target1.ext', 'path/to/target2.ext'],
      source = 'path/to/source.ext',
      file_cmd = "command",
      params = 'arguments or options',
      depends = ['path/to/dependency1.ext', 'path/to/dependency2.ext']
)
```

The additional options are as follows:

- `file_cmd` is the command that SCons should pass to Stata's batch mode. The default is `do`, but the user can specify anything that Stata can accept as a command, for example, `dyndoc` or `markdown`.
- `params` are arguments or options that should follow the `source` in the call to Stata batch mode. For example, a call to `markdown` might specify

  `params = ', saving(myfile.html) replace'`
- `depends` is an alternative to using the `Depends()` function.

We include several additional examples of these options in our web tutorial.[19]

18. For managing tasks that do not build specific targets, see section 3.3.5.
19. See https://bquistorff.github.io/statacons/swc.html, especially the *Parameters* lesson.

3.3.3 Defining lists of files in SConstructs

As projects grow, it may become unwieldy to write out long lists of dependencies. Furthermore, if different targets have overlapping sets of dependencies, writing the same dependencies in different places is inefficient and can lead to errors. We can do better by defining variables in our SConscript files by using standard Python syntax and some special SCons functions. As a preview of the applied example in section 4, we illustrate two of these methods.

The target `ctyQtrWages.dta` uses as inputs five Excel files: `wagedata1.xlsx`, ..., `wagedata5.xlsx`. We could write these out explicitly in the `Depends()` line, but instead we will demonstrate two alternatives. The first is the SCons `Glob()` function:

```
wageDataFiles = Glob("inputs/wagedata[1-5].xlsx")
Depends(wage, wageDataFiles)
```

where `[1-5]` means "any element of the sequence 1, 2, ..., 5."[20] SCons will search the directory `inputs` for files matching the pattern and will define `wageDataFiles` to be the set of files that matches.

A more flexible alternative is to use Python's "list comprehension" method:[21]

```
wageDataFiles = ["inputs/wagedata"+e+".xlsx"
                 for e in ['1','2','3','4','5']]
Depends(wage, wageDataFiles)
```

While this method is not as concise as `Glob()`, it has a few advantages. First, it allows for more complex structures and patterns. For example, a sequence of ISO country codes would begin `'AFG','ALA','ALB'`. Second, it can define lists of targets that may not yet exist, while `Glob()` is limited to matching existing files. Third, it is more precise. As a concrete example, suppose the file `wagedata1.xlsx` is missing from the `dataprep/inputs` directory. If we write out all our dependencies explicitly or use the list comprehension method above, SCons will return an error, which is useful. If we use `Glob("inputs/wagedata[1-5].xlsx")`, SCons will think that the dependencies are `wagedata2.xlsx`, ..., `wagedata5.xlsx`—that is, the files that are present—and will proceed with the build.[22]

See the companion web tutorial[23] for more examples of using Python coding to write more concise `SConstructs`.

20. `Glob()` allows Unix shell pattern matching, that is, `*` (match everything), `?` (match a single character), `[seq]` (match any single character in the sequence `seq`), and `!seq` (match any single character not in the sequence `seq`). See the *SCons User Guide* (SCons Development Team 2021b) for more details.
21. See https://docs.python.org/3/tutorial/datastructures.html for more information on list comprehension and other Python data structures.
22. Despite our concerns about `Glob()`, we have used it in our applied example (section 4) for the sake of providing an example. See the definition of the `industryCompFiles` variable in `dataprep/SConstruct`.
23. See https://bquistorff.github.io/statacons/swc.html, especially the *Variables* lesson.

3.3.4 SConstruct functions

We discuss only a few of the more useful functions available in `SConstruct` files. See the SCons manual (SCons Development Team 2021a) or the *SCons User Guide* (SCons Development Team 2021b) for a comprehensive list and more details on implementation. We also note a few additional options that the `pystatacons` package provides.

Some of these functions (for example, `SetOption()`), will require the use of Unix-like standard SCons syntax rather than our adaptation to Stata syntax.

`AlwaysBuild(`*target*[`,` *target*`...`]`)` instructs SCons to always rebuild the given targets when they are specified, even if they are up to date. A target must be specified either in the command line or as part of the default set—that is, `AlwaysBuild()` neither implies nor is implied by `Default()`.

`Default()` allows the user to specify which targets SCons will examine by default. In our earlier example, we could set a default of rebuilding only `auto-modified.dta` but not the other targets by adding

```
Default('outputs/auto-modified.dta')
```

to the `SConstruct`. By default, SCons will examine all targets in the current directory or subdirectories unless otherwise specified on the command line.

`Decider(`*function*`)` determines how SCons will decide whether a target is up-to-date.

`"content"` is the default. For each dependency of a target, SCons will calculate a signature and compare the result against the last time the target was built.[24] This means that a target will be rebuilt only if one of its dependencies has changed. So suppose `A.dta` is a dependency of `B.dta` and `B.dta` is a dependency of `C.dta`. Now suppose that `A.dta` changes, but when `B.dta` is rebuilt, it does not change. Even though `B.dta` is newer than `C.dta`, SCons will not rebuild `C.dta` because its dependency did not change.

`"timestamp-newer"` requests that SCons rebuild a target if any of its dependencies are newer than the target itself (similarly to the classic build tool `make`). Suppose `A.dta` is a dependency of `B.dta` and `B.dta` is a dependency of `C.dta`. Now suppose that `A.dta` changes, but when `B.dta` is rebuilt, it does not change. Even though `B.dta` has not changed, it is newer than `C.dta`, so SCons will rebuild `C.dta`. Because it does take time for SCons to calculate signatures, `"timestamp-newer"` may be faster despite some unnecessary rebuilds if there are relatively many input files (and therefore many signatures to calculate) and relatively low computation time.

24. Rather than recalculating the signature, SCons will save time by using a cached value of the signature if (1) a cached value is available from previous builds, (2) the modification timestamp has not changed, and (3) the timestamp is at least as old in seconds as the `max-drift` parameter (default is two days but is configurable).

`"timestamp-match"` requests that SCons rebuild a target if any of its dependencies have a timestamp different (newer or older) from the last time the target was built. If all dependencies' timestamps are the same as the last time the target was built, the target will be considered up-to-date and will not be rebuilt.

`"content-timestamp"` requests that SCons rebuild a target only if a dependency's timestamp and signature have changed since the last time the target was rebuilt, after which the dependency is considered up-to-date. This is similar to `content`, except it skips checking signatures for dependencies whose timestamps are unchanged and therefore may run faster.

`"content-timestamp-newer"` requests that SCons rebuild a target only if a dependency's signature has changed since the last time the target was rebuilt and any of its dependencies are newer than the target itself. Similarly to the `assume_done()` and `assume_built()` command-line options defined above, this means that SCons will not rerun Stata code that has already been run outside of SCons. Unlike `assume_done()` and `assume_built()`, it only needs to be specified once in the `SConstruct`, not specified at the command line each time. `"content-timestamp-newer"` is defined by `pystatacons`. To use it, include `Decider(pystatacons.decider_str_lookup["content-timestamp-newer"])` in your `SConstruct` after `pystatacons` has been imported.

`Ignore(`*target*`,` *dependency*`)` tells SCons to ignore the file *dependency* when deciding whether to rebuild *target*.

`Ignore(`*directory*`,` *target*`)` tells SCons to remove *target* from the set of default targets. The first argument of `Ignore()` must be the *directory* where *target* would be rebuilt. SCons will still build *target* if 1) *target* is specified on the command line or 2) *target* is a dependency of another target and needs to be rebuilt.

`Requires(`*target*`,` *prerequisite*`)` specifies that *target* be built after *prerequisite*, even if *prerequisite* is not a dependency for *target*. This is useful if the user wishes to impose some order on the build even if that order is not required by the build logic. For example, the user may want to move longer-running tasks later in the build so she can inspect the output of shorter-running tasks while the others run.

`SetOption(`*name*`,` *value*`)` allows some command-line options to be set in the `SConstruct`. For example, `SetOption('silent',1)` is equivalent to `silent` on the command line. See the *SCons User Guide* (SCons Development Team 2021b) for the options that can be set by `SetOption()`. The command line overrides values set through `SetOption()`.

3.3.5 Alias and phony targets

The `Alias()` function allows users to give names to targets or groups of targets. This can make it easier to build only a subset of targets or to refer to groups of targets in an `SConstruct`.

In our introductory example in section 2, suppose we wanted a shorthand for the data prep process so that, rather than call `outputs/auto-modified.dta` from the command line, we could just use `dataprep`, which is shorter and easier to remember. We would code

```
dataprep_Targets = ['outputs/auto-modified.dta']
Alias('dataprep',dataprep_Targets)
```

into our `SConstruct` and then, if we wanted to build only the data prep targets, call

```
statacons dataprep
```

instead of

```
statacons outputs/auto-modified.dta
```

For additional uses of `Alias()` to create a convenient way to reference groups of files, see `SConstructWithAlias` and other `SConstruct`s from the section 2 example.[25]

A second use of `Alias()` is to incorporate commands associated with a project that do not build files: configuring the local environment, communicating with a server (for example, publishing a package), or displaying information in the terminal. For convenience, these can be stored in your `SConstruct` by assigning them to phony targets (targets that are not real files) by using `Alias()`. For example, if we put the following line in our `SConstruct`, we could run the shell command `dir` whenever we run SCons with the target `show_dir`.

```
env.AlwaysBuild(env.Alias("show_dir", action="dir"))
```

Because `show_dir` is not a real target with an associated build recipe, we encase `Alias()` in `AlwaysBuild()`. The command `dir` could be replaced with a Python function (potentially user-defined) to allow for more options, as we will see in section 6.1.3.

3.4 Default settings and configuration files

We have incorporated what we think are sensible defaults into `statacons`. These include the following:

- finding the Stata executable automatically by searching the user's `PATH` variable and other default locations;
- using our custom `complete_datasignature.ado` to compute signatures for `.dta` that depend only on their content, not their embedded timestamp (see discussion in section 5); and
- deleting the batch-mode generated log files after a do-file is successfully executed in batch mode, but retaining it in the default directory if the do-file fails.

25. https://github.com/bquistorff/statacons/raw/main/examples/stataconsIntro.zip

By setting these defaults, we reduce the time users need to figure out how to configure the tool, which we suspect is a barrier to adoption of build tools. However, some users may wish to choose other options or have unusual setups. Therefore, we allow users to change their configurations in two files, `config_project.ini` (for settings common to all users in a given project) and `config_local.ini` (for user-specific settings). We provide templates for both with suggested values in the project files installed with `statacons`. In case of conflicts between the two files, `config_local.ini` will take precedence.

These configuration files follow a simplified INI format readable by Python's standard library.[26] They are simple text files that are easily read and edited. Each file stores key-value pairs organized under one or more headers.

```
[header1]
key1: value 1
key2: value 2
[header2]
key3: value 3
```

As an example, the user may wish to retain the log files created by batch-mode Stata in a folder called `logs` and detail the specific Stata executable to use, which can be done in `config_local.ini` with

```
[SCons]
success_batch_log_dir: ./logs/
[Programs]
stata_exe: "C:/Program Files/Stata17/StataMP-64.exe"
```

As a second example, by default SCons operates relative to the directory where the `SConstruct` is found, so this is the directory from which Stata will run in batch mode. This is a reasonable default, but some users may wish for Stata to start in a different directory. We allow this through a `stata_chdir` option in `config_project.ini`. See the `config_project.ini` file provided with this package for the available options.

We provide some additional examples of the use of configuration files elsewhere in this article, such as telling SCons where to find shared folders in Dropbox. Users can specify other configuration files by using the `config_file()` option listed in section 3.2.2, *Custom statacons options*. As mentioned there, in the case of conflicts between config files, the file listed later in `config_file()` will take precedence.

3.5 Other advanced features

We conclude this section by briefly mentioning a few advanced features of SCons. This is not a complete list nor a comprehensive guide; rather, we aim to familiarize readers with some concepts and vocabulary they may encounter while using the tool.

26. See https://docs.python.org/3/library/configparser.html for full details.

The first is the `SConscript`, which allows users to split a long `SConstruct` into more manageable parts, that is, a hierarchical build; see online appendix A.1.

The second is SCons's ability to manage parallel builds, conducting tasks simultaneously rather than sequentially when the logic of the workflow permits. SCons parallel builds are well suited for a relatively small number of relatively slow tasks. For tasks with a relatively high number of relatively quick tasks, like bootstrapping or simulation, the approach of `parallel` (Vega Yon and Quistorff 2019) is preferable. We provide an application of parallel builds to our applied example in online appendix A.2. Parallel builds are currently only available in SCons (that is, from a terminal), not in `statacons`.

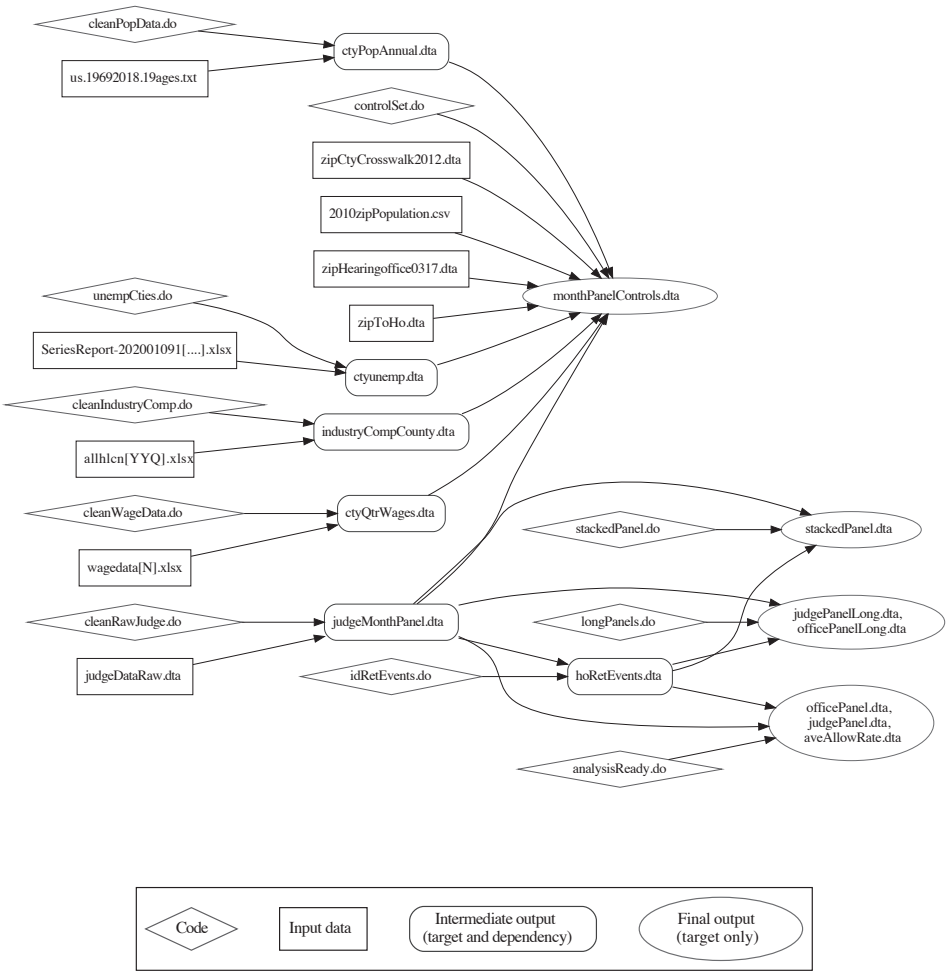
The third is the SCons cache, which allows storing and sharing of derived files. We describe the use of the cache in section 7.2 in the context of collaborative workflows.

4 Applied example

This section describes the use of `statacons` in an empirical project (Shumway and Wilson 2022). Key build scripts and do-files are provided in `appliedExample.zip`, posted to our repository.[27]

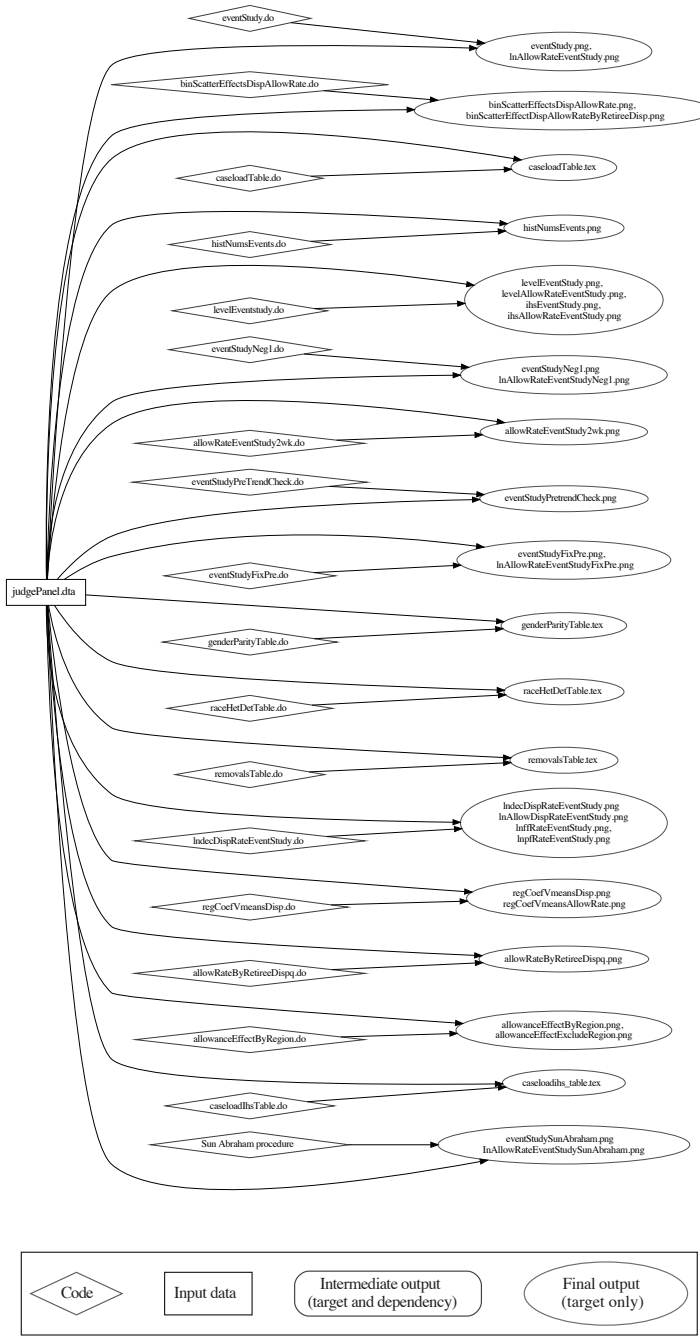
For our purposes, this project has two main parts: data prep, in which raw data are cleaned and merged and new variables are created for analysis, and analysis, in which the analysis of these data is carried out. The value of a build tool is apparent from figures 3, 4, 5, and 6: rather than trying to remember all of these interdependencies, we encode them in `SConstruct`s. Consulting an `SConstruct` and the output of `tree()` is a more reliable way to document interdependencies, orient a new collaborator, remind the user's future self of how a project works after some time away, trace errors, and so on, than depending on memory. The `SConstruct` and output of `tree()` for both the data prep and the analysis tasks are provided in `appliedExample.zip` on the project repository.

27. https://github.com/bquistorff/statacons/raw/main/examples/appliedExample.zip



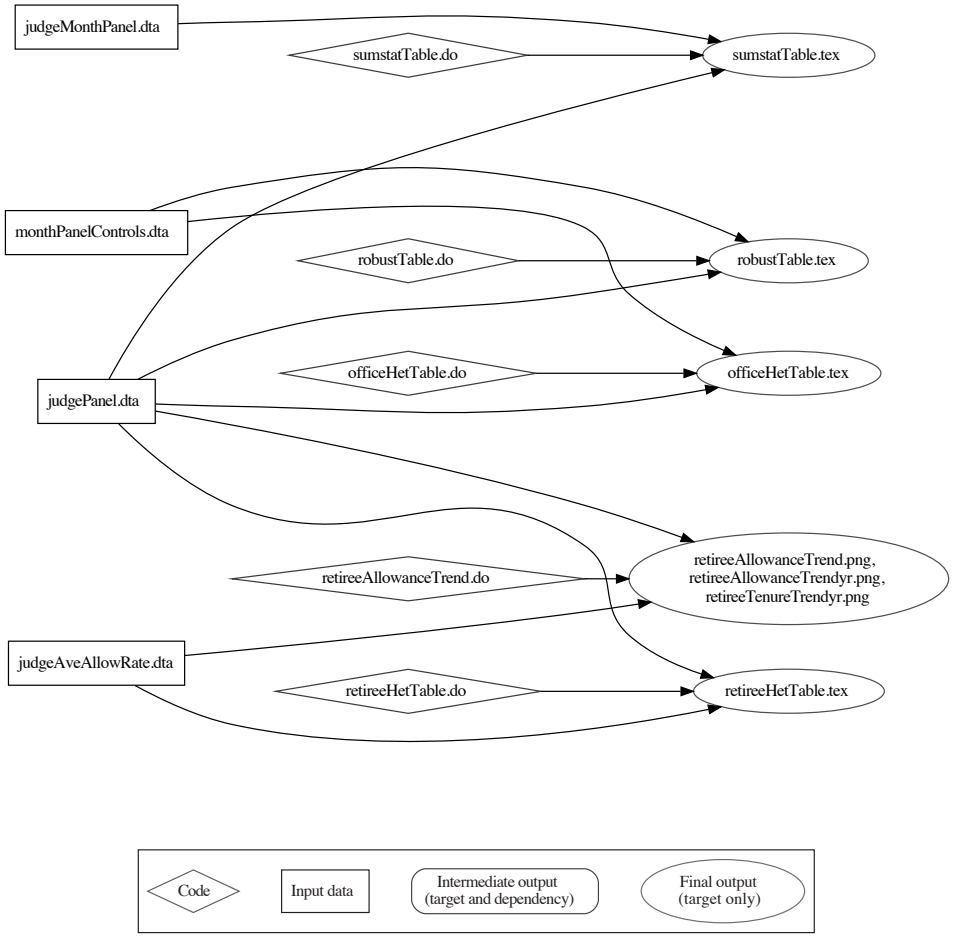
NOTES: Brackets represent a series of similarly named files; for example, `wagedata[N].xlsx` represents `wagedata1.xlsx`, ..., `wagedata5.xlsx`. We have omitted file paths for brevity.

Figure 3. Workflow for applied example: Data prep



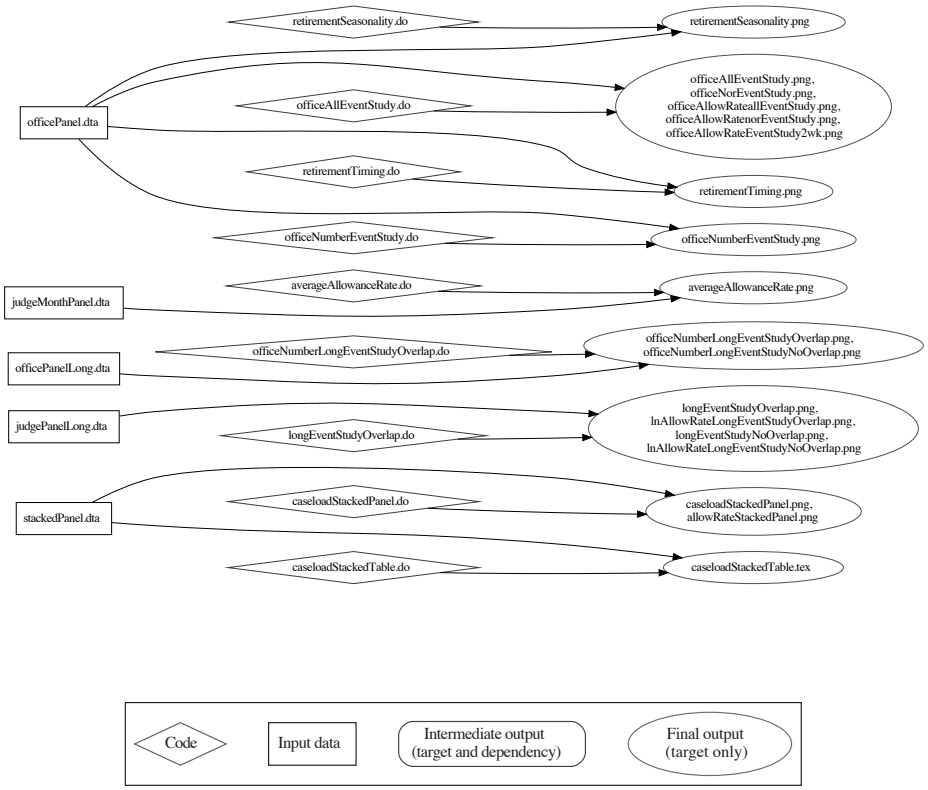
NOTES: We have omitted file paths for brevity. The workflow for the Sun–Abraham event study procedure is simplified; see figure 7 for the complete workflow.

Figure 4. Workflow for applied example: Analysis (part 1)



NOTE: We have omitted file paths for brevity.

Figure 5. Workflow for applied example: Analysis (part 2)



NOTE: We have omitted file paths for brevity.

Figure 6. Workflow for applied example: Analysis (part 3)

We have separated the two main parts of the project, data cleaning and data analysis, into two directories, each with its own `SConstruct`. These two directories are linked by a third, called `transferDataprep`. The phony target `post_outputs` in the data prep `SConstruct` copies the main outputs from `dataprep/outputs` to `transferDataprep`, while the phony target `pull_inputs` in the analysis `SConstruct` copies these from `transferDataprep` to `analysis/inputs`. See section 6.1.2 for more details and an extension to shared files.

One of the estimation procedures, the dynamic treatment-effects estimator of Sun and Abraham (2021), takes over two hours to run, even on a fast desktop. We have written our code and `SConstruct` to avoid repeating this estimation when it is not necessary. The relevant section of the `SConstruct` is shown in figure 7 along with a diagram.

```
# Sun-Abraham event study estimation methodology
sunAbrahamSelect = env.StataBuild(
   source = "code/sunAbrahamSelect.do",
    target = ['outputs/dta/sunAbrahamEstimationSample.dta']
)
Depends(sunAbrahamSelect, ['inputs/judgePanel.dta'])
# do estimation
sunAbrahamEst = env.StataBuild(
    source = "code/sunAbrahamEstimation.do",
    target = ['outputs/dta/sunAbrahamResults.dta']
)
Depends(sunAbrahamEst, ['outputs/dta/sunAbrahamEstimationSample.dta'])
# plot results
sunAbrahamFigs = env.StataBuild(
    source = "code/sunAbrahamGraph.do",
    target = ['outputs/figures/eventStudySunAbraham.png',
        'outputs/figures/lnAllowRateEventStudySunAbraham.png']
)
Depends(sunAbrahamFigs, ['outputs/dta/sunAbrahamResults.dta'])
```

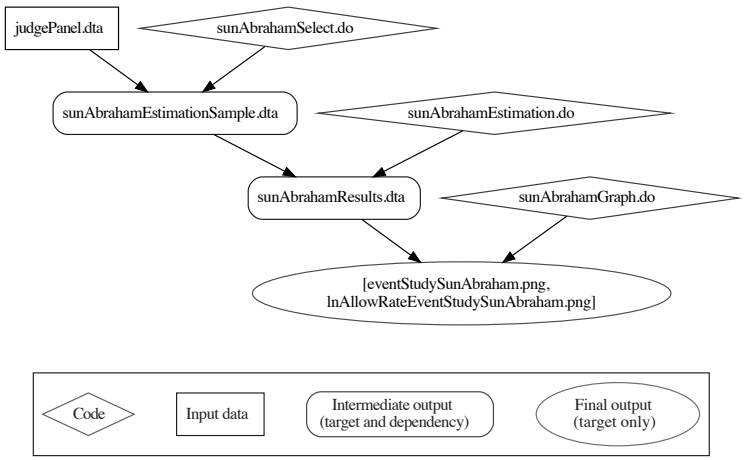


Figure 7. Separation of concerns in applied example

This procedure uses a subset of the variables and observations in `judgePanel.dta`. While we do need to rerun the estimation if the data used have changed, we would not want to rerun the estimation if the changed variables or observations were not used in this task. The task `sunAbrahamSelect` creates `sunAbrahamEstimationSample.dta`, which keeps only the variables and observations of `judgePanel.dta` needed for this task. If `judgePanel.dta` changes, then SCons rebuilds `sunAbrahamEstimationSample.dta`. If `sunAbrahamEstimationSample.dta` changes, then SCons continues to the estimation step in `sunAbrahamEst`, because `sunAbrahamEstimationSample.dta` is the dependency for that task. However, if `sunAbrahamEstimationSample.dta` has not changed, then SCons will not repeat `sunAbrahamEst`.

Similarly, the ultimate outputs are two graphs of estimated treatment effects. Suppose we wanted to change some aspect of the graphs, for example, the axis titles. We would not want to have to repeat the full estimation. On the other hand, we do want to re-create these graphs whenever the underlying estimates change. We handle this by separating the estimation and the creation of graphs into distinct steps. The task `sunAbrahamEst` handles the estimation, saving the estimation results in `sunAbrahamResults.dta`.[28] Then the task `sunAbrahamFigs` handles producing the graphs. If we edit `sunAbrahamGraphs.do` to change some aspect of the graphs, SCons will not see any need to rebuild `sunAbrahamEst`. On the other hand, if something about the estimation changes such that `sunAbrahamResults.dta` changes, SCons will automatically rebuild the graphs.

This division of tasks is known as separation of concerns in computer science and is, in our view, a key distinction between working with a build tool and working in the literate programming paradigm described in section 6.2.2. The virtue of literate programming is that everything is in one piece of code, which is certainly convenient and simple. A build tool workflow in which discrete tasks are separated will save computation time. Of course, there is a cost to creating separate do-files, so this tradeoff may not be worthwhile for simple, quick tasks.

Finally, we can further reduce computation time by using SCons's ability to manage parallel builds. We discuss this in the online appendix A.2.

5 Technical details

We note here some technical details of how our package works.

First, as we have discussed above, SCons's default behavior is to look for changes in the content of dependencies when deciding whether to rebuild a target. SCons does this by computing a signature of each dependency. However, because Stata embeds a timestamp in `.dta` files, `.dta` files created at two different times will have different signatures, even if the contents are identical. This will lead to unnecessary rebuilds of target files. Stata's `datasignature` command can create signatures that depend only on the values of the data in the `.dta` file, not on the timestamp, but `datasignature` signatures do not depend on variable labels or value labels.

28. Here we have saved estimation results into a dataset. An alternative is to use Stata's `estimates save` command to save results to a `.ster` file. The community-contributed `estwrite` and `estread` (Jann 2005) improve on `estimates save` by allowing users to store multiple estimation results in a single `.ster` file and then refer to individual estimates by model name. As of version 1.2.5, `estwrite` has option `reproducible` that will eliminate the timestamps previously embedded in `.ster` files. As of this writing, `.ster` files produced by `estimates save` contain an embedded timestamp.

We create an ado-file called `complete_datasignature.ado` that augments Stata's `datasignature` with a signature of the dataset's metadata (variable formats, variable and value labels, notes and characteristics) but not the timestamp. We then patch SCons so that it uses our `complete_datasignature.ado` instead of its default signature algorithm. In the `config_project.ini` file, the user has several options:

- `Strict` will include data and all metadata in the signature (variable formats, variable and value labels, and notes and characteristics).
- `LabelsFormatsOnly` will include data, variable formats, and variable and value labels in the signature (but will not include dataset labels and notes and characteristics).
- `DataOnly` will use Stata's standard `datasignature`, which depends on data only (and does not depend on any metadata or the embedded timestamp).
- `False` will use SCons's default signature, which will depend on the embedded timestamp.

In addition, we provide some control over the use of `datasignature`'s `fast` mode. The `fast` mode speeds up the calculation of the file signature but is not machine-independent; that is, users on different machines may produce different signatures. This is not so important for solo users working on a single machine. But for collaborative workflows (or a single user working on different machines) that save time by sharing built files in a common directory, using the `fast` option may lead `statacons` to incorrectly believe that a build is out of date. We make `fast` the default unless the `CacheDir()` option is specified (indicating a collaborative workflow; see section 7.2), in which case the machine-independent, slower method is the default. Users can override these defaults—for example, to use the slow option for a solo project—through the `dta_sig_mode` option in the `config_project.ini` configuration file.

Our next set of issues deals with SCons directing Stata to run in batch mode. First, SCons needs to know if the do-file finished successfully to know if targets should be considered built. This would typically be communicated via the program's exit status, but Stata always returns a normal exit, even if the script it runs encountered an error. Our Stata builder, therefore, parses the end of the Stata batch-mode log file to see if there was an error and communicates this back to SCons. This requires that batch-mode Stata produce plain-text log files, so we do not allow the user to change the options with which batch-mode Stata is called.

Second, Stata typically uses the do-file name as the name of the batch-mode log file, but we may need to change this for a variety of reasons. If two commands would result in the same log filename (for example, we call the same do-file with different parameters or we have multiple non-`do` commands), then they cannot be run in parallel because Stata will fail to start when the second instance cannot open the log file. Therefore, for all commands aside from plain `do` commands (with no parameters), we have Stata run a generated do-file that then runs the user's command. We name this generated file so

that its resulting log file will be unique (we take the original command's typical log file name and append a short hash of the text of the command).

Finally, although SCons is a Python tool, and Stata 16+ has a Python environment, several hurdles need to be overcome to use SCons from inside Stata. First, Stata sets a handler for several program "signals" for Python at startup. Later, SCons temporarily sets new signal handlers. When SCons is done, though, it tries to reinstate the original handlers in a way that does not deal correctly with the handlers Stata set from outside Python. We therefore patch the SCons function that reinstates the signal handlers to deal with this case and avoid a crash. Second, Stata changes the `stdout` and `stderr` communication channels for the Python process at startup. Later, SCons temporarily changes these but does not restore them correctly when finished. We patch this restore function so that output can continue to be displayed in the Stata window. Third, Stata's Python environment is kept intact between Python script calls. SCons, however, is typically only able to be called from a clean environment. We therefore remove references to SCons modules before starting another run.

6 Extensions, alternative approaches, and limitations

6.1 Extensions

There are several ways to add to the basic functions we have described to this point. In increasing order of difficulty, these are to use additional build tools already supplied with SCons; use built-in SCons commands; execute Python commands and utilities; add community-contributed SCons build tools; or write a new builder. We provide a few examples of the first three and provide references for those interested in the last two.

6.1.1 Additional built-in build tools: Compiling LaTeX with SCons

SCons comes with a builder for LaTeX documents. As long as the user has a TeX distribution (for example, TexLive or MiKTeX) installed and known to the system, this will work basically out of the box. Conveniently, SCons will automatically scan the `.tex` file for dependencies, so it is not necessary to code dependencies explicitly into the `SConstruct`. By default, these implicit dependencies include `.tex` files that have been included or imported (via `\include{}` or `\imported{}`), graphics files (`.pdf`, `.eps`, etc.) included through `\includegraphics{}`, and `.bib` files, among others. The user may need to adjust some construction variables to get the document to compile in the desired way.

All tools built into SCons are listed in the *SCons User Guide* (SCons Development Team 2021b).[29]

29. As of Version 4.3.0 (December 2021), they are listed in *Appendix C. Tools*, https://scons.org/doc/production/HTML/scons-user/apc.html.

6.1.2 Built-in SCons commands: Copy built files across directories

SCons has several built-in utilities that can be convenient tools. See sections 11 and 12 of the *SCons User Guide* (SCons Development Team 2021b) for a full list. One simple and especially useful tool, `Install()`, copies files across directories. We use `Install()` extensively in the code for this article.[30]

First, in section 2, we use `Install()` to post our table, graph, and compiled PDF to Overleaf for use in this article. We need to tell `statacons` where it should copy these files, that is, to the Overleaf directory. In general, Overleaf files will reside in `Dropbox/apps/Overleaf/ProjectName`, where in this case our project name is `SJ-BuildTool`. If there is only one user, or if all users have the same path to Dropbox,[31] we can define this directly in the `SConstruct`. For a Windows user, this would be

```
# define Overleaf directory
overleaf_dir = 'C:/Users/UserName/Dropbox/apps/Overleaf/SJ-BuildTool'
```

If there are multiple users with different paths to Dropbox, each path must[32] be defined in the user's own `config_local.ini` configuration file:

```
# define Overleaf directory
overleaf_dir: C:/Users/UserName/Dropbox/apps/Overleaf/SJ-BuildTool
```

Then, in the `SConstruct`, we read in this variable:

```
# read in overleaf directory from config_local.ini
overleaf_dir = env['CONFIG']['Project']['overleaf_dir']
```

Having obtained the target location, we can add our `Install()` command to the `SConstruct`:

```
# transfer files to overleaf
env.Install(overleaf_dir, [cmd_analysis, pdf_output])
env.Alias('post_to_overleaf', overleaf_dir)
```

`Install()` takes two arguments: the first is the directory to which the files should be copied and the second is the list of files to be copied. Here the first argument is the path to the Overleaf folder, as defined in the `overleaf_dir` variable. The second argument, `[cmd_analysis, pdf_output]`, tells SCons to copy the targets of the tasks `cmd_analysis` and `pdf_output`. Finally, in the last line, we use `Alias()` to give this task a convenient name. Now we can post our output files to Overleaf with

```
statacons post_to_overleaf
```

30. All code is available on our project website, and the project files included with `statacons` contain templates for the discussed configuration files, with extensive examples.
31. For example, through a symbolic link.
32. Adept Python coders may be able to program around this in the `SConstruct` by accessing the operating system environment. As we note in section 7, we encourage users who develop enhancements to `statacons` to make them available publicly, and we provide a project Wiki to host.

Our second application of `Install()` is in section 4, our applied example.[33] The two components to this project are data prep and analysis. We want the outputs of data prep to become inputs for analysis. A single user working alone could simply write code in analysis to use output files from data prep directly. Or the user could write a single `Install()` task in the data prep `SConstruct` to push from `dataprep/outputs` to `analysis/inputs` or in the analysis `SConstruct` to pull from `dataprep/outputs` to `analysis/inputs`. These strategies, however, may not work well in a collaborative project, because a user in analysis would not want datasets to be changed without warning, and a user in data prep would not want work in progress to be pulled into analysis. At the cost of slightly increasing complication, we set up a more deliberate workflow.

First, we add a `transferDataPrep` directory on the same level as the `dataprep` and `analysis` directories.

Second, we add an `Install()` task to the data prep `SConstruct` that the user can select to push finished outputs from `dataprep/outputs` to `transferDataPrep`.

```
# post outputs to ../transferDataPrep folder
env.Install('../transferDataPrep', outputs)
env.Alias('postOutputs', '../transferDataPrep')
```

To avoid transferring incomplete work, we want the `postOutputs` task to run only when it is explicitly called (that is, via `statacons postOutputs` at the Stata prompt or `scons postOutputs` in a terminal). In this case, `postOutputs` does not run automatically because its target, the `transferDataPrep` directory, is not in the `dataprep` directory; by default, SCons will look for targets only inside the `SConstruct`'s directory and subdirectories.

However, if the target directory is in `dataprep` or its subdirectories, we will need to remove `postOutputs` from the default targets. There are two ways to do this. The first is to set the `Default()` function explicitly and not include the transfer task among the defaults. This may be cumbersome if there are many targets. The second way is to use the `Ignore()` function in the `SConstruct` to tell SCons not to build `postOutputs` unless explicitly called from the command line. The code in the `SConstruct` would be

```
# post outputs to ./transferDataPrep folder
env.Install('./transferDataPrep', outputs)
env.Alias('postOutputs', './transferDataPrep')
# because ./transferDataPrep is a subdirectory, need to
# Ignore postOutputs so that it is not built by default
Ignore(postOutputs,'./transferDataPrep')
```

Third and finally, we set up a similar task in the analysis `SConstruct` to pull data from `transferDataPrep` into `analysis/inputs`. In this example, we demonstrate the use of the `shutil.copytree`[34] Python utility:

33. https://github.com/bquistorff/statacons/raw/main/examples/appliedExample.zip
34. The `dirs_exist_ok` option of `shutil.copytree` requires Python 3.8 or higher.

```
# get inputs from folder "judgesExample/transferDataPrep"
# relative path from judgesExample/analysis is ../transferDataPrep
# need to specify alias pullInputs at command line
def copy_inputs(src = "../transferDataPrep", dst="inputs", **kwargs):
    import shutil
    shutil.copytree(src, dst, dirs_exist_ok=1)
env.AlwaysBuild(env.Alias("pullInputs", action=copy_inputs))
```

While we have made the `transferDataPrep` directory local in this example, it could be a shared folder, such as on Dropbox or Google Drive. Each user would just have to specify the correct path in their `config_local.ini` configuration file.

6.1.3 Python commands: Download and unzip archives

It is straightforward to use Python utilities in SCons. As an example, the data for our applied example in section 4 are stored on Dropbox in a zip file. In `dataprep/SConstruct`, we include instructions to download and unzip the `.zip` file by using Python utilities.

```
def url_download(fname = "judgesExampleInputs.zip",
      url="https://dl.dropboxusercontent.com/s/abc123/judgesExampleInputs.zip",
      **kwargs):
    import urllib.request
    urllib.request.urlretrieve(url, fname)
env.AlwaysBuild(env.Alias("dl_orig_data", action=url_download))

env.AlwaysBuild(env.Alias("unzip_inputs",
    action="python -m zipfile -e judgeExampleInputs.zip .")
)
```

Because these actions do not have targets, their aliases must be specified in the call to `statacons`, for example,

```
statacons dl_original_data
statacons unzip_inputs
```

6.1.4 Other types of extensions

There are many other ways to add to SCons. For example, many community-contributed tools are implemented as Python packages.[35] Also, the user can write commands, builders, or tools; see the *SCons User Guide* (SCons Development Team 2021b) for documentation.[36] A helpful step-by-step example is provided by Wichman (2021).

35. See https://github.com/SCons/scons/wiki/ToolsIndex for a list.
36. As of version 4.3.0 (December 2021), the relevant chapters are 17, *Extending SCons: Writing Your Own Builder*, and 18, *Not Writing a Builder: The Command Builder*.

6.2 Alternative approaches

6.2.1 Alternative build tools

We chose to work in SCons for two main reasons. First, because it is a Python tool, we could integrate it into Stata without requiring the use of a shell. Second, it provides the fundamental desirable features of build systems, as articulated in Mokhov, Mitchell, and Peyton Jones (2018): it is "minimal" in that it only rebuilds when dependencies have changed and only runs each task once, and it provides "early-cutoff optimization," knowing when a newly built object has not changed and then ceasing to rebuild downstream targets. There are many alternatives to SCons; in this section, we list several and discuss what we see as the advantages and disadvantages relative to SCons.

The best-known build tool is `make`. SCons has a few advantages over `make`. First, `make` rebuilds targets when any dependency's timestamp is newer than the target, which leads to unnecessary rebuilds if a dependency's content has not changed.[37] Second, `make` uses a custom syntax that is typically difficult for beginners (it uses many special characters and has unforgiving rules regarding which whitespace characters to use). Third, `make` requires the use of a Unix shell (or emulator, like Terminal on Mac or Windows Subsystem for Linux) to be used interactively (using `shell make` in Stata will not show you the output). One advantage of `make` over SCons is that it is somewhat easier to code straightforward commands into a script without having to create a build environment. A second advantage is the `makefile2graph` utility, which produces graphs of dependency trees; we are not aware of a similar utility for SCons.

There are many other build tools, such as `cmake`, some of which have support for using content signatures instead of file timestamps, but most are not in Python and so cannot be used interactively inside of Stata. Among Python-based build tools, we chose to work in SCons because its widespread use means there are active discussion and support communities.

The community-contributed `project` command (Picard 2013) in Stata is an ingenious pure Stata approach: dependencies are encoded into a project's do-files, and the `project` program maintains a database of signatures to decide which outputs to rebuild. We prefer working in SCons for a few reasons, including integration with other tools, configurability, and access to advanced options, such as the cache and parallel jobs. We also like that SCons does not require making changes to existing do-files.

37. SCons can replicate this behavior with the `Decider()` function mentioned in section 3.3.4.

6.2.2 Literate programming

A more popular approach to reproducible research in Stata has followed the literate programming paradigm, in which text and code are combined in a single document. There are several excellent options in this paradigm, including built-in Stata commands such as `dyndoc` and `putdocx`, "notebooks" like the Jupyter notebooks popular in Python and now accessible in Stata 17, and community-contributed commands like `texdoc` (Jann 2016), `webdoc` (Jann 2017), `markdoc` (Haghish 2016), and `markstat` (Rodríguez 2017).

The literate programming approach is appealing and is certainly useful in many cases.[38] A literate programming script does provide a degree of self-documentation because input and output filenames are present in the same file as the text. However, literate programming has limitations similar to those of a master do-file: as projects grow more complex and computationally intensive, either all analyses must be repeated every time the document is produced or the user must make some manual decisions about which parts to refresh and which to skip. (See our discussion of separation of concerns in section 4.) Fortunately, a build tool such as `statacons` can complement a literate programming tool by automating some of these decisions. For example, our web tutorial consists of several webpages, each produced by a corresponding `dyndoc` file. We have written an `SConstruct` to manage this process so that we rebuild a webpage if, and only if, the inputs have changed. This code is available on our Wiki page.[39]

6.3 Limitations

The main limitation we have found with SCons is that SCons only recognizes builds it has performed itself. That is, SCons determines which targets need to be rebuilt by comparing the current state of the project to its state after the last time a project was built by SCons, as recorded in the `.sconsign.dblite` database.[40]

As an example, suppose that in our introductory example in section 2, the user has edited `analysis.do` to change an axis title in the target `scatterplot.pdf`. The user runs code from the Stata Command window in the usual way (`do code/analysis.do`) and is satisfied. In reality, the targets `scatterplot.pdf` and `regressionTable.tex` are now up to date. However, SCons will not realize this. SCons will compute a signature of `analysis.do`, see that it has changed since its signature was last recorded in `.sconsign.dblite`, and believe that the targets need to be rebuilt.

In `make`, if the user knows that a build is up-to-date, the `--touch` option will change targets' timestamps to the present. Because `make` rebuilds only when dependencies are newer than the target, `make` will no longer rebuild the target.

38. For example, we used `texdoc` with `statacons` to produce section 2 of this article, `dyndoc` with `statacons` to produce the companion tutorial website, and `markdoc` to produce our help files.
39. https://github.com/bquistorff/statacons/wiki/Statacons-and-literate-programming
40. This applies for all settings of the `Decider()` function, including `timestamp-newer`.

SCons does not have a similar option, so we have added the `assume_built()` and `assume_done()` options to the `statacons` command, in which the user can designate certain targets that do not need to be rebuilt or do-files that do not need to be rerun. If the already built targets all have newer timestamps, then our "content-timestamp-newer" `Decider()` can also be used. See the respective entries in sections 3.2.2 and 3.3.4.

A second, related limitation is that setting up a collaborative project—for example, keeping code under version control (for example, Git) and sharing files through a service like Dropbox—requires a bit of care. Our preferred approach is a shared SCons cache. This is a powerful tool provided by SCons, but there is a learning curve. We describe this method and some alternatives in sections 7.2 and 7.3.

Finally, as mentioned in section 6.2, it is easier to implement one-off tasks with `make` than with SCons.[41] We try to minimize this limitation by providing a few examples that users can adapt, but some familiarity with Python coding and some trial-and-error will likely be required to extend these examples to other tasks.

7 Collaborative workflows with statacons

In this section, we discuss using statacons with collaborators. On our project Wiki, we provide a simple worked example of a collaborative project using the SCons cache along with GitHub, which is our preferred approach.[42]

7.1 The problem

The main difficulty with collaboration is that, as we mentioned in section 6.3, SCons only understands builds it has done itself. That is, SCons compares the current state of targets and dependencies to their state as recorded in the `.sconsign.dblite` database the last time SCons ran the build.

As an example, suppose that users A and B are collaborating on the introductory example of section 2. User A is working primarily on the data prep task, and user B is working primarily on analysis. Imagine for the sake of this example that the data prep task takes a very long time to run. User A has made some edits to `dataprep.do` and updated `auto-modified.dta`. User A shares these two files with user B.

The issue is that user B's SCons will look at `dataprep.do`, compare its current status to its status the last time `auto-modified.dta` was built as recorded in user B's `.sconsign.dblite` database, see that `dataprep.do` is different, and conclude that `auto-modified.dta` needs to be rebuilt. User B could use the added `assume_built()` or `assume_done()` options to tell SCons that in fact `auto-modified.dta` is up-to-date, but in more complex projects, this sort of mental dependency-tracking is prone to failure.

41. At least, it is easier if the user is already comfortable using a shell and shell commands.
42. https://github.com/bquistorff/statacons/wiki/Collaboration-using-GitHub-and-the-SCons-cache

The underlying problem is that there are three competing values: we want builds to be complete (all targets up-to-date), we do not want to duplicate work, and we want to automate the build and avoid manual workarounds, which are error-prone. Our preferred approach is to use the SCons cache to share built files. While this requires a small amount of effort to set up and some care to maintain, it satisfies all three values without introducing too much additional complexity. We will describe this approach in the next section, and then mention a few alternatives and their limitations.

7.2 SCons cache

Our preferred approach is to use the SCons cache to share derived files. See listing 4 for an example `SConstruct`. In the `SConstruct`, we designate a shared folder to store the cache (`CacheDir('path/to/cache')`):

```
# **** Setup from pystatacons package *****
import pystatacons
env = pystatacons.init_env()

# use an sconsign database specific to this SConstruct
SConsignFile(".sconsignCache")

# set path to cache directory
# shared cache dir
# hard-coded -- if hard-coding path with spaces, *must* enclose in quotes
# dropbox example
#CacheDir('C:/Users/Username/Dropbox/SJ-BuildTools/simpleCacheShare')
# local folder example
#CacheDir('./scons_cache')
# from config_local.ini -- in config, do *not* enclose path with spaces in quotes
CacheDir(env['CONFIG']['Project']['cache_dir'])

# **** Substance begins        *****

# analysis
cmd_analysis = env.StataBuild(
    target = ['outputs/scatterplot.pdf',
              'outputs/regressionTable.tex'],
    source = 'code/analysis.do'
)
Depends(cmd_analysis, ['outputs/auto-modified.dta'])

# dataprep
cmd_dataprep = env.StataBuild(
    target = ['outputs/auto-modified.dta'],
    source = 'code/dataprep.do'
)
Depends(cmd_dataprep, ['inputs/auto-original.dta'])
```

Listing 4. `SConstruct` for introductory example using SCons cache

Rather than hard code the path into the `SConstruct`, the path can be coded into a configuration file (for example, `config_project.ini`, `config_local.ini`, or another file specified through the `config_file()` option), which the `SConstruct` can then read. Here we use a local `scons_cache` folder so that this can be a self-contained example, but in practice, this may be a shared drive or a folder on a file-sharing service. See `config_local_template.ini` for several examples.

SCons will then store built files in the cache and, before building targets, check to see whether an up-to-date version is available in the cache. See the *SCons User Guide* (SCons Development Team 2021b) for details on how this works.[43]

We have found that this approach handles the three competing goals of a build tool well (complete builds; reduce unnecessary builds; automation instead of manual intervention), with a few caveats.

The main caveat is that, by default, SCons will update the cache whenever a target is newly built. This is not desirable while writing and testing code because it will lead to filling the cache with unnecessary files, and other users will be repeatedly downloading these unnecessary files. It is good practice to update the cache only when you are sure the work is complete. There are two ways to do this.

The first way is to use the command-line option `cache_readonly` to tell SCons to check and read from the cache but not update the cache with any built files or use `cache_disable` to tell SCons to ignore the cache entirely. Once you have successfully completed your build, then use the option `cache_force` to force SCons to update the cache. This will also make it less likely that two users will attempt to update the same cached file at the same time.

The second way is to edit the `SConstruct` so that the default is to have the cache disabled and a command-line option is required to read or write to the cache. We provide an example on our project Wiki.[44]

There are a few additional caveats. First, the cache can get large and periodically need to be cleaned and rebuilt.[45] Second, when using the cache, SCons calculates build signatures regardless of the `Decider()` method chosen, so there are no time savings from using, for example, `content-timestamp`. Of course, this does not affect those using the default `content`. Finally, in its own storage, the cache uses file signatures as filenames, instead of the cached file's own name, which is slightly inconvenient if you need to find and examine the cached version of a particular file. The `cache_debug` option will provide information on the files being used. Note that the file will be stored in a subfolder corresponding to the first two characters of the cached filename; for example, a cached file `d7581eba1ac59ad5f92b2fceba04477f` will be in subfolder `d7`.

We list some useful command-line options for the cache below. Using these options together with `dry_run` can be instructive to get a preview of how SCons will interact with the cache.

43. As of SCons 4.3.0 (December 2021), details are in chapter 22, *Caching Built Files*. See especially the first footnote in that chapter, which describes the information SCons stores and uses to determine whether a target is up-to-date.
44. https://github.com/bquistorff/statacons/wiki/Cache-with-cache-disabled-as-default
45. Rebuilding the cache does not require rebuilding the project, just erasing the cache and then calling SCons with the option `cache_force`.

We list the Stata-style syntax first, and then the standard SCons syntax. Recall that either syntax is allowed but the two cannot be mixed in a single command.

`cache_debug(-)` prints information on the cache files being used to the screen.

SCons equivalent: `--cache-debug=-`

`cache_debug(`*file*`)` saves information on the cache files being used to the file *file*.

SCons equivalent: `--cache-debug=`*file*

`cache_disable` specifies to not use files from the cache and not write files to the cache.

SCons equivalents: `--cache-disable`, `--no-cache`

`cache_force` writes all derived files to the cache, whether they are built in this call to SCons or have previously been built. This overrides the default, which is to write only files that are built or rebuilt in the current call to SCons.

SCons equivalents: `--cache-force`, `--cache-populate`

`cache_readonly` specifies to use files from the cache but not write files to the cache.

SCons equivalent: `--cache-readonly`

`cache_show` requests that SCons print what it would have done to build the file if it had not used the cache. By default, SCons prints "`retrieved` *file* `from cache`" to the screen when it pulls a file from the cache rather than building it.

SCons equivalent: `--cache-show`

7.3 Other options

Here are a few alternatives, along with what we see as their main drawbacks.

7.3.1 Shared .sconsign.dblite database

Because SCons uses the `.sconsign.dblite` database to decide which targets need to be rebuilt, it is tempting to use a shared database, whether through a file-synchronization service like Dropbox or version-control system like Git or GitHub. However, this is unlikely to work well. Because the `.sconsign.dblite` database is a single, binary file, version conflicts or file corruption can arise even if users are working on different aspects of a project. Furthermore, unless all users have all of their code, targets, and dependencies synchronized at all times, a shared `.sconsign.dblite` database will cause SCons to get very confused, because it will see discrepancies between files and database entries every time any user updates any file. This might be workable if all users are synced to the same shared folder, for example, a shared Dropbox folder, but this is not possible if one user is using more full-featured source code and collaboration management solutions, such as Git.

7.3.2 Marking files as built

The `assume_built()` and `assume_done()` options we have added for `statacons` can help in one-off cases; see section 3.2.2 for their descriptions. In the example in section 7.1, user B could update to user A's latest `dataprep.do` and `auto-modified.dta`, and then run

```
statacons, assume_built('outputs/auto-modified.dta')
```

SCons will then skip rebuilding `auto-modified.dta`, record the current file signature as in `.sconsign.dblite`, and move on to rebuilding the final outputs.

While this approach can be useful, it relies on users accurately keeping track of what is up-to-date and making manual choices in builds, both tasks that build tools are intended to make unnecessary. For example, we have found `assume_built(*)` to be useful if a minor change to the builder has caused SCons to think all files need to be rebuilt when in fact none of them do. Again, though, we have to be sure that the change to the builder will not result in any substantive change.

7.4 Additional comments on collaborative workflows

7.4.1 Version control

If you are using a version control system like Git or SVN, we recommend that you do not keep `config_local.ini` or `.sconsign.dblite` under version control. In the case of `config_local.ini`, this is simply because the purpose of this file is to account for things that will vary across users. In the case of `.sconsign.dblite`, see our discussion in section 7.3.1.

There are a few additional Python-generated files (for example, `__pycache__`) that should not be kept under version control. In Git, this is handled with a `.gitignore` file. In the replication archive we provide for our introductory example in section 2, we have included an example `.gitignore` that excludes the files above. In addition, we recommend excluding all generated files. That is, as a rule, we keep only code (in the broad sense of things, that which a human has typed, as opposed to that which has been generated by the computer) under version control. This is generally considered good practice but is essential when using the SCons cache, because the cache is already essentially handling version control for generated files, and having both SCons and the version control software attempting the same task will lead to conflicts.

We do recommend keeping community-contributed programs under version control to ensure that all users have the same version of these programs. In the introductory example, we kept the community-contributed `estout` command (Jann 2007) in the folder `code/ado/plus`. Our `profile.do` sets Stata's `adopath` to look in this folder (and subfolders). Similarly, we keep `config_project.ini`, the `SConstruct`, `statacons.ado`, and the other associated files under version control, again so that the build will be consistent across users. The downside of this practice is that you will need to maintain separate copies of the same programs for each project.

7.4.2 Shared SCons cache

If you are using Dropbox or a similar service to share your SCons cache, make sure that you have offline access to the folder containing the cache.[46] Using the "Smart Sync" or "Streaming Access" options that store files online and then fetch them when requested is likely to cause several problems. First, Python may not find folders that are only available in the Cloud. Second, this will slow down the process of retrieving cached files. Third, you may not be able to access certain recently cached files when working offline.

Finally, as of April 2022, we do not recommend Google Drive for sharing caches. Google Drive will sometimes be able to discern the type of cached files and will add a file extension to the filename in the cache. This happens especially frequently with PDFs. This will confuse SCons and prevent it from successfully retrieving that file from the cache. We have not found this to occur with Dropbox.

8 Conclusion

In this paper, we have provided introductions to build tools and `statacons`, a build tool for Stata that we have adapted from SCons. `statacons` runs directly in Stata without an external shell, can be used with a minimum of initial configuration, and does not require modification of existing code. While there is a startup cost to integrating `statacons` into a project, and it requires some discipline to keep `SConstruct`s up-to-date, the benefits—reliable reproducibility and saved computation time—are considerable.

We recommend that users start small. While retrofitting a large, existing project may prove frustrating for new users, using `statacons` and a build-tool mindset from the beginning of a new project can promote better code by encouraging breaking down tasks into parts, simplifying the scope of individual pieces of code, and separation of concerns. In debugging problems, starting small is also advised: First, run code in Stata to see if the problem is with the code rather than the build tool. Then, build one or a few targets at a time until the problem is isolated. And finally, examine the information

46. As of April 2022, in Dropbox, either turn off "Smart Sync" to make all your Dropbox content available offline, or use "Selective Sync" to keep your cache folders available offline. While we currently do not recommend Google Drive for shared caches, if you do use Google Drive, choose either "Mirror files" to make all your Google Drive content available offline, or follow the instructions under "Stream files—Choose specific files and folders to make available offline" to keep your cache folders available offline.

provided by the `debug`, `tree`, and `show_config` options as well as the `stataconsign` function.[47]

We hope that this paper will encourage the use of build tools in Stata. One barrier is that there are few worked examples that are simple enough for the novice to understand easily but also show some of the power of the tool. We provide a few such examples in this paper and the accompanying material. We hope that other users will build on this work by providing additional annotated examples and by extending the scope of tasks that can be performed. We encourage users to share these advances publicly, and we provide a project Wiki to host such contributions.

9 Acknowledgments

We thank Riley Wilson for generously agreeing to our request to use Shumway and Wilson (2022) as the applied example in this article. We thank the editor, Prof. Stephen P. Jenkins, and an anonymous referee for comments that improved the article.

Quistorff is the corresponding author, though for concerns regarding the package (additional information, questions, suggestions, and bug reporting) please use our project repository: https://github.com/bquistorff/statacons/.

The authors are listed alphabetically. All views expressed in this article are our own and do not necessarily reflect those of our employers or any other institutions with which we are affiliated.

47. The do-file `debugging_checklist.do`, included with the `statacons` package, may be useful.

10 Program and supplemental materials

For the latest version of the software and installation instructions, go to the project webpage: https://bquistorff.github.io/statacons.

For previous versions of the software and installation instructions, go to our project archive: https://osf.io/gbh4m/.

11 References

BITSS. 2020. Guide for accelerating computational reproducibility in the social sciences. https://bitss.github.io/ACRE.

Christensen, G., Z. Wang, E. L. Paluck, N. Swanson, D. J. Birke, E. Miguel, and R. Littman. 2019. Open science practices are on the rise: The state of social science (3S) survey. Technical report, MetaArXiv. https://doi.org/10.31222/osf.io/5rksu.

Gentzkow, M., and J. M. Shapiro. 2014. Code and data for the social sciences: A practitioner's guide. Mimeo: University of Chicago. http://web.stanford.edu/~gentzkow/research/CodeAndData.pdf.

Haghish, E. F. 2016. markdoc: Literate programming in Stata. *Stata Journal* 16: 964–988. https://doi.org/10.1177/1536867X1601600409.

Jackson, M. 2016. Software carpentry: Automation and make. http://swcarpentry.github.io/make-novice/.

Jann, B. 2005. estwrite: Stata module to store estimation results on disk. Statistical Software Components S450201, Department of Economics, Boston College. https://ideas.repec.org/c/boc/bocode/s450201.html.

———. 2007. Making regression tables simplified. *Stata Journal* 7: 227–244. https://doi.org/10.1177/1536867X0700700207.

———. 2014. Software Updates: st0085_2: Making regression tables from stored estimates. *Stata Journal* 14: 451. https://doi.org/10.1177/1536867X1401400217.

———. 2016. Creating LaTeX documents from within Stata using texdoc. *Stata Journal* 16: 245–263. https://doi.org/10.1177/1536867X1601600201.

———. 2017. Creating HTML or Markdown documents from within Stata using webdoc. *Stata Journal* 17: 3–38. https://doi.org/10.1177/1536867X1701700102.

Mokhov, A., N. Mitchell, and S. Peyton Jones. 2018. Build systems à la carte. *Proceedings of the ACM on Programming Languages* 2: 1–29. https://doi.org/10.1145/3236774.

Orozco, V., C. Bontemps, E. Maigné, V. Piguet, A. Hofstetter, A. Lacroix, F. Levert, and J.-M. Rousselle. 2020. How to make a pie: Reproducible research for empirical

economics and econometrics. *Journal of Economic Surveys* 34: 1134–1169. https://doi.org/10.1111/joes.12389.

Picard, R. 2013. project: Stata module providing a set of tools to build and manage a Stata project. Statistical Software Components S457685, Department of Economics, Boston College. https://ideas.repec.org/c/boc/bocode/s457685.html.

Rodríguez, G. 2017. Literate data analysis with Stata and Markdown. *Stata Journal* 17: 600–618. https://doi.org/10.1177/1536867X1701700304.

SCons Development Team. 2021a. SCons 4.3.0 Man page. https://scons.org/doc/4.3.0/PDF/scons-man.pdf.

———. 2021b. SCons 4.3.0 User Guide. https://scons.org/doc/4.3.0/PDF/scons-user.pdf.

Shumway, C., and R. Wilson. 2022. Workplace disruptions, judge caseloads, and judge decisions: Evidence from SSA judicial corps retirements. *Journal of Public Economics* 205: 104573. https://doi.org/10.1016/j.jpubeco.2021.104573.

Stodden, V., J. Seiler, and Z. Ma. 2018. An empirical analysis of journal policy effectiveness for computational reproducibility. *Proceedings of the National Academy of Sciences* 115: 2584–2589. https://doi.org/10.1073/pnas.1708290115.

Sun, L., and S. Abraham. 2021. Estimating dynamic treatment effects in event studies with heterogeneous treatment effects. *Journal of Econometrics* 225: 175–199. https://doi.org/10.1016/j.jeconom.2020.09.006.

Vega Yon, G. G., and B. Quistorff. 2019. parallel: A command for parallel computing. *Stata Journal* 19: 667–684. https://doi.org/10.1177/1536867X19874242.

Wichman, M. 2021. ToolsForFools. GitHub. https://github.com/SCons/scons/wiki/ToolsForFools.

Wilson, G., D. A. Aruliah, C. T. Brown, N. P. Chue Hong, M. Davis, R. T. Guy, S. H. D. Haddock, et al. 2014. Best practices for scientific computing. *PLOS BIOLOGY* 12: e1001745. https://doi.org/10.1371/journal.pbio.1001745.

About the authors

Raymond P. Guiteras is an assistant professor in the Department of Agricultural and Resource Economics and is core faculty in the Global WaSH Research Cluster at North Carolina State University.

Ahnjeong Kim is a PhD student in economics at North Carolina State University.

Brian Quistorff is a research economist in the Office of the Chief Economist at the Bureau of Economic Analysis.

Clayson Shumway is a PhD student in economics at North Carolina State University.

Installation guide

Requirements

These installation instructions correspond to `statacons` v3.0.0, released with this version of the article. For up-to-date installation instructions, corresponding to the latest version of `statacons`, see our project webpage: https://bquistorff.github.io/statacons.

The following are required for `statacons`:

- Stata 16.0 or later (any edition)
- Python 3.5 or later[48]
- SCons 4.2 or later

Step 1: Python setup

We provide the basic steps for installing Python and configuring Stata for use with Python. For more details, we recommend a series of posts by Chuck Huber on the Stata Blog.[49]

Python commands are issued at a command prompt in a terminal window. Exactly which terminal will depend on your operating system and implementation of Python. Some popular examples include

- Windows
  - Python: Windows Command Prompt ("`cmd.exe`") or Powershell
  - Anaconda: Anaconda Powershell Prompt or Anaconda Prompt
- Mac OS: Terminal
- Unix-like (for example, Linux): Terminal

Step 1.1: Install Python

Install Python 3.5 or later (3.8 or later recommended).

48. Some of the advanced options in our applied example in section 4 require Python 3.8 or later.
49. Available at https://blog.stata.com/tag/python/; see especially posts 1 and 3, *Setting up Stata to use Python* and *How to install Python packages*.

Step 1.2: Configure Stata to work with Python

Step 1.2.1: Locate Python executable

In Stata, type

```
python search
```

to locate the Python executable. If your system has both Python 2 and Python 3 installed, be sure to use the Python 3 executable.

Step 1.2.2: Set path to Python executable in Stata

Stata will use the default Python unless configured before first use. To do so, in Stata type

```
python set exec pyexecutable , permanently
```

where *pyexecutable* is a path found in step 1.2.1.

If Stata is started in a specific Anaconda environment, we provide a simple command in Stata to query the environment and set the Python path accordingly:

```
set_python_exec_env
```

Step 1.2.3: Check setup

In Stata, check your setup by typing

```
python query
```

Step 1.3: Install Python packages scons and pystatacons

`pip` is the standard way to install Python packages. In the terminal, type

```
pip install scons
pip install pystatacons
```

To check that these are installed and recognized by Stata, open Stata and type

```
python which SCons      // case-sensitive
python which pystatacons
```

Step 2: statacons installation and setup

Step 2.1: Install statacons package

First, check your `adopath`. By default, `statacons` will be installed to your `PLUS` directory. Users working in collaboration with others may wish to change the `PLUS` and `PERSONAL` directories to be local to the project so that all users will be using the same

version of community-contributed do-files. See the `profile_template.do` we have included in our project files for what we recommend.

Second, `statacons` requires both the core program files and a key set of project files. You may wish to install these separately. To install the main program, type

```
net install statacons, from(https://raw.github.com/bquistorff/statacons/main/)
```

The core program files are

- `complete_datasignature.ado`
- `statacons.ado`
- `stataconsign.ado`
- `runscons.py`
- `sconsign-script.py`
- `sconstruct_fns.py`

To check that `statacons` is installed, type `which statacons` in Stata.

Step 2.2: Install statacons project files

To install the project files, navigate to the root of your project directory and then run

```
// change to project's root directory
cd path/to/your/project
// set other to current directory, that is, project's root directory
net set other .
// get project files
net get statacons, from(https://raw.github.com/bquistorff/statacons/main/)
unzipfile project_files
rm project_files.zip
```

The project files packages in `project_files.zip` are

- `SConstruct`
- `config_project.ini`
- `utils/config_local_template.ini`
- `utils/profile_template.do`
- `utils/debugging-checklist_template.do`
- `utils/.gitignore_template`

The only project file required is `SConstruct`. You may also wish to use the template files; if so, copy them into the main project folder (and remove the `_template` suffix).

Step 2.3: Configuration

`statacons` is intended to work without additional user configuration, for example, by automatically detecting the location of the Stata executable. However, some configuration may be required. For example, if you have more than one version of Stata installed, `statacons` may not find your preferred version first.

Our default is to use two configuration files, `config_project.ini`, which sets parameters that are common to all users working on a given project, and `config_local.ini`, which sets parameters specific to the local user. The files `config_project.ini` and `config_local_template.ini` provided with the Stata project files give several examples of how to set commonly used parameters. See section 3.4 for more information on these configuration files and how to read from them in `SConstruct`s.

The default is that these configuration files, if used, will be stored in the same directory as the `SConstruct`. The command line option `config_file()` can instruct SCons to use a custom location or filename instead; see section 3.2.2.

To check the configuration of `statacons`, in Stata type `statacons, show_config`. The do-file `debugging-checklist.do` included in the `utils` folder may also be useful in checking that components are in the right places.

Step 2.4 (optional): Setup for version control

If you are using version control software, such as Git, here are a few recommendations:

- Set your ado-path so that `statacons` and other packages are installed in the project folder. This will ensure that all users have the same version of all ado-files. See `profile_template.do` in the `utils` folder.
- Keep the following files under version control: `SConstruct` and `config_project.ini`.
- Exclude the following files from version control: `config_local.ini`, all generated files (including SConsign files, for example, `.sconsign.dblite`), and batch-generated `.log` files. See `.gitignore_template` in the `utils` folder for what we recommend.

See section 7.4.1 for more details on working with version control.

Step 3: Test run with introductory example

Replication code and data for our introductory example in section 2, as well as the extensions to that example, are provided in `stataconsIntro.zip`, posted to our repository.[50] To run the example, unpack the zip archive and follow the steps in section 2.

50. https://github.com/bquistorff/statacons/raw/main/examples/stataconsIntro.zip