Creating a Project Using an Existing Build System

In this section:

Introduction

To automatically create a project using an existing build system, C/C++test requires information about the build process of an existing code base. To provide C/C++test with necessary data, you can use the `cpptestscan` or `cpptesttrace` utility shipped with C/C++test to create a C/C++test project that you would normally build using build tools such as GNU make, CMake, or QMake. The utilities utilities output a build data file that includes build information required by C/C++test (see About Build Data Files (.bdf) for details). Alternatively, you can first build a project and then configure it manually using the information collected by the utilities.

If you use CMake, you can define a C/C++test project directly in the CMake build file. This allows you to automatically generate the C/C++test project during the build with CMake – without having to run `cpptestscan` or `cpptesttrace`. See Integrating C/C++test into a CMake Build for details.

About Build Data Files (.bdf)

Build information, such as the working directory, command line options for the compilation, and link processes of the original build, are stored in a file called the build data file. The following example is a fragment from a build data file:

```
------- cpptestscan v. 9.4.x.x -------
working_dir=/home/place/project/hypnos/pscom
project_name=pscom
arg=g++
arg=-c
arg=src/io/Path.cc
arg=-Iinclude
arg=-I.
arg=-o
arg=/home/place/project/hypnos/product/pscom/shared/io/Path.o
```

The build data file can be used as a source of information about project source files, compiler executable, compiler options, linker executable, and options used to build the project. There are three ways to use the build data file to create a project:

- Manually setting up 'Use options from the build data file' as the options source for the project and selecting appropriate build data file (see Creating a Project from the GUI).
- Using the GUI to automatically import a project. See Importing project using Build Data File with the GUI wizard.
- Using the command line to automatically import a project. See Importing a Project from the Command Line.

> **Note**
>
> Required environment variables can also be stored in the build data file if the following apply:
>
> - Your build system sets up the required environment variables for the compiler / linker to work correctly
> - These variables are not available in the environment when run C++tests.
>
> See description of the `'--cpptestscanEnvInOutput'` option below.

Using `cpptestscan` or `cpptesttrace` to Create a Build Data File

The `cpptestscan` and `cpptesttrace` executables are located in the C++test installation directory. They collect information from the build process of an existing code base, generate build data files with the information, and append information about each execution into a file.

The `cpptestscan` utility is used as a wrapper for the compiler and/or linker during the normal build. To use `cpptestscan` with an existing build, build the code base with `cpptestscan` as the prefix for the compiler / linker executable of an existing build to build the code base. This can be done in two ways:

- Modify the build command line to use `cpptestscan` as the wrapper for the compiler/linker executables
- If you don't want to (or cannot) override the compiler variable on the command line, embed `cpptestscan` in the actual make file or build script.

To use `cpptesttrace` with an existing build, build the code base with `cpptesttrace` as the prefix for the entire build command. `cpptesttrace` will trace the compiler and linker processes executed during the build and store them in the build data file.

In both cases, you need to either add the C++test installation directory to your PATH environment variable, or specify the full path to either utility.

Additional options for `cpptestscan` and `cpptesttrace` are summarized in the following table. Options can be set directly for the `cpptestscan` command or via environment variables. Most options can be applied to `cpptestscan` or `cpptesttrace` by changing the prefix in command line.

Basic `cpptestscan` usage:

- Windows: `cpptestscan.exe [options] [compile/link command]`
- Linux: `cpptestscan [options] [compile/link command]`

Basic `cpptesttrace` usage:

- Windows: `cpptesttrace.exe [options] [build command]`
- Linux: `cpptesttrace [options] [build command]`

Option	Environment Variable	Description	Default

`--cpptestscanOutputFile=<OUTPUT_FILE>`
 	
`--cpptesttraceOutputFile=<OUTPUT_FILE>`
	CPPTEST_SCAN_OUTPUT_FILE)	Defines file to append build information to.	cpptestscan.bdf

`--cpptestscanProjectName=<PROJECT_NAME>`
 	
`--cpptesttraceProjectName=<PROJECT_NAME>`
	CPPTEST_SCAN_PROJECT_NAME	Defines suggested name of the C++test project.	name of the current working directory

`--cpptestscanRunOrigCmd=[yes|no]`
 	
`--cpptesttraceRunOrigCmd=[yes|no]`
	CPPTEST_SCAN_RUN_ORIG_CMD	If set to "yes", original command line will be executed.	yes

`--cpptestscanQuoteCmdLineMode=[all|sq|none]`
 	
`--cpptesttraceQuoteCmdLineMode=[all|sq|none]`
	CPPTEST_SCAN_QUOTE_CMD_LINE_MODE	Determines the way C++test quotes parameters when preparing cmd line to run.
		
		`all`: all params will be quoted
		
		`none`: no params will be quoted
		
		`sq`: only params with space or quote character will be quoted
		
		cpptestscanQuoteCm dLineMode is not supported on Linux
			all

`--cpptestscanCmdLinePrefix=<PREFIX>` `--cpptesttraceCmdLinePrefix=<PREFIX>`	CPPTEST_SCAN_CMD_LINE_PREFIX	If non-empty and running original executable is turned on, the specified command will be prefixed to the original command line.	[empty]
`--cpptestscanEnvInOutput=[yes|no]` `--cpptesttraceEnvInOutput=[yes|no]`	CPPTEST_SCAN_ENV_IN_OUTPUT	Enabling dumps the selected environment variables and the command line arguments that outputs the file. For advanced settings use – cpptestscanEnvFile and – cpptestscanEnvars options	no
`--cpptestscanEnvFile=<ENV_FILE>` `--cpptesttraceEnvFile=<ENV_FILE>`	CPPTEST_SCAN_ENV_FILE	If enabled, the specified file keeps common environment variables for all build commands; the main output file will only keep differences. Use this option to reduce the size of the main output file. Use this option with – cpptestscanEnvInOut put enabled	[empty]
`--cpptestscanEnvars=[*|<ENVAR_NAME>,...]` `--cpptesttraceEnvars=[*|<ENVAR_NAME>,...]`	CPPTEST_SCAN_ENVARS	Selects the names of environment variables to be dumped or '*' to select them all. Use this option with – cpptestscanEnvInOut put enabled.	*
`--cpptestscanUseVariable=[VAR_NAME=VALUE,...]` `--cpptesttraceUseVariable=[VAR_NAME=VALUE,...]`	CPPTEST_SCAN_USE_VARIABLE	Replaces each occurence of "VALUE" string in the scanned build information with the "${VAR_NAME}" variable usage.	[empty]
`--cpptesttraceTraceCommand`	CPPTEST_SCAN_TRACE_COMMAND	Defines the command names that will be traced when collecting build process information. These names, specified as regular expressions, should match original compiler / linker commands used in the build process.	

Example: Modifying GNU Make Build Command to Using cpptestscan

Assuming that a make-based build in which the compiler variable is CXX and the original compiler is g++:

```
      make -f </path/to/makefile> <make target> [user-specific options] CXX="cpptestscan --cpptestscanOutputFile=/path/to/name.bdf --cpptestscanProjectName=<projectname> g++"
```

This will build the code as usual, as well as generate a build data file (name.bdf) in the specified directory.

Note

When the build runs in multiple directories:

- If you do not specify output file, then each source build directory will have its own .bdf file. This is good for creating one project per source directory.
- If you want a single project per source tree, then a single .bdf file needs to be specified, as shown in the above example.

Example: Modifying GNU Make Build Command Using cpptesttrace

Assume that a regular make-based build is executed with:

```
make clean all
```

you could use the following command line:

```
cpptesttrace --cpptesttraceOutputFile=/path/to/name.bdf --cpptesttraceProjectName=<projectname> make clean
all
```

This will build the code as usual and generate a build data file (name.bdf) in the specified directory.

Note

If the compiler and/or linker executable names do not match default `cpptesttrace` command patterns, they you will need to use `--cpptesttraceTraceCommand` option described below to customize them. Default cpptestscan command trace patterns can be seen by running 'cpptesttrace --cpptesttraceHelp' command.

Example: Modifying GNU Makefile to use cpptestscan

If your Makefile uses CXX as a variable for the compiler executable and is normally defined as CXX=g++, you can redefine the variable:

```
ifeq ($(BUILD_MODE), PARASOFT_CPPTEST)
CXX="/usr/local/parasoft/cpptestscan --cpptestscanOutputFile=<selected_location>/MyProject.bdf --
cpptestscanProjectName=MyProject g++"
else
CXX=g++
endif
```

Next, run the build as usual and specify an additional BUILD_MODE variable for make:

```
make BUILD_MODE=PARASOFT_CPPTEST
```

The code will be built and a build data file (MyProject.bdf) will be created. The generated build data file can then be used to create a project from the GUI or from the command line.

Note

The `cpptestscan` and `cpptesttrace` utilities can be used in the parallel build systems where multiple compiler executions can be done concurrently. When preparing Build Data File on the multicore machine, for example, you can pass the `-j <number_of_parallel_jobs>` parameter to the GNU make command to build your project and quickly prepare the Build Data File.

The following examples demonstrate how to create a .bdf file for CMake projects using `cpptestscan` or `cpptesttrace`. For complex scenarios and unit testing, consider integrating C/C++test into a CMake build using the C/C++test extension for CMake. See Integrating C/C++test into a CMake Build.

Example: Using cpptesttrace with CMake build

Assuming that you have a CMake-based build, you can produce a build data file using cpptesttrace:

1. Run the original CMake command to use CMake to generate make files. For example:

   ```
   cmake -G "Unix Makefiles" ../project_root
   ```

2. Setup environment variables for `cpptestscan` making sure to use an absolute path for the output file:

```
export CPPTEST_SCAN_PROJECT_NAME=my_project
export CPPTEST_SCAN_OUTPUT_FILE=$PROJ_ROOT/cpptestscan.bdf
```

3. Make sure the `cpptesttrace` executable is available on the PATH.
4. Run the project build normally but with `cpptesttrace` as a wrapper. For example if normal build command is make clean all for the build with 'cpptesttrace' the command will be `cpptesttrace make clean all`

A build data file will be generated in the location defined by the CPPTEST_SCAN_OUTPUT_FILE variable. If the varialbe is isn't set, the build data file(s) will be generated in the Makefiles' locations.

Example: Using cpptestscan with CMake build

All scripts and commands are bash-based – adapt them as needed for different shells.

Assuming a CMake-based build, do the following to produce a build data file using cpptestscan:

1. Use CMake to re-generate make files with the 'cpptestscan' used as a compiler prefix. Make sure the 'cpptestscan' executable is available on the PATH.
    a. If original CMake command is `cmake -G "Unix Makefiles" ../project_root`, then you need to get rid of existing CMake cache and run cmake overriding compiler variables. The following example assumes 'gcc' is used as a C compiler and 'g++' as a C++ compiler executable:

    ```
    rm CMakeCache.txt
    CC="cpptestscan gcc" CXX="cpptestscan g++" cmake -G "Unix Makefiles" ../project_root
    ```

    b. Look in the CMakeCache.txt file to see if CMAKE_*_COMPILER variables point to cpptestscan.
    c. If the make files are re-generated, jump to step 5. Continue if `cmake` failed in the boot-strap phase because the compiler wasn't recognized.
2. Prepare the `cpptestscan` wrapper scripts that will behave like a CMake compiler by creating the following BASH scripts. In this example, we assume 'gcc' is used as a C compiler and 'g++' as a C++ compiler executable:

```
>cat cpptest_gcc.sh
#!/bin/bash
cpptestscan gcc --cpptestscanRunOrigCmd=no $* > /dev/null 2>&1 gcc $*
exit $?

>cat cpptest_g++.sh
#!/bin/bash
cpptestscan g++ --cpptestscanRunOrigCmd=no $* > /dev/null 2>&1 g++ $*
exit $?
```

    - The first script invokes `cpptestscan` to extract options without running the compiler. The second script runs the actual compiler so that the entire script looks and acts like a compiler in order to be "accepted" by CMake.
3. Give the scripts executable attributes and place them in a common location so they are acces-sible to everyone who needs to scan make files. Make sure the `cpptestscan` and the scripts are available on the PATH.
4. Use CMake to re-generate make files with the scripts used as compilers by extending the original CMake command with options that re-generate make files from the prepared scripts.
    - Original CMake command is `cmake -G "Unix Makefiles" ../project_root` then you need to get rid of existing CMake cache and run cmake overriding compiler vari-ables. In the following example, we assume 'gcc' is used as a C compiler and 'g++' as a C++ compiler executable:

    ```
    rm CMakeCache.txt
    cmake -G "Unix Makefiles" -D CMAKE_C_COMPILER=cpptest_gcc.sh -D CMAKE_CXX_COMPILER=cpptest_g++.sh
    ../project_root
    ```

    - Look in the CMakeCache.txt file to see if CMAKE_*_COMPILER variables point to prepared wrappers.
5. Setup environment variables for `cpptestscan`; make sure to use an absolute path for the out-put BDF file:

```
export CPPTEST_SCAN_PROJECT_NAME=my_project
export CPPTEST_SCAN_OUTPUT_FILE=$PROJ_ROOT/cpptestscan.bdf
```

6. Run the project build normally without overwriting any make variables. Build data files(s) will be generated in location defined by the CPPTEST_SCAN_OUTPUT_FILE variable or - if not set -in location of Makefiles.

Note

By default CMake-generated make files only print information about performed actions without actual compiler/linker command lines. Add "VERBOSE=1" to the make command line to see executed compiler/linker command lines.

Using cpptestscan or cpptestrace with other Build Systems

For non-make based build systems, usage of cpptestscan and cpptesttrace is very similar to the examples shown above. Typically, a compiler is defined as a variable somewhere in the build scripts. To create a Build Data File from that build system using cpptestscan, prefix the original compiler executable with cpptestscan. To create a Build Data File from that build system using cpptesttrace, prefix whole build command line with cpptesttrace.

When should I use cpptestscan?

It is highly recommended that the procedures to prepare a build data file are integrated with the build system. In this way, generating the build data file can be done when the normal build is performed without additional actions.

To achieve this, prefix your compiler and linker executables with the cpptestscan utility in your Makefiles / build scripts.

When should I use cpptesttrace?

Use cpptesttrace as the prefix for the whole build command when modifying your Makefiles / build scripts isn't possible or when prefixing your compiler / linker executables from the build command line is too complex.

Importing Project Using Build Data File with the GUI Wizard

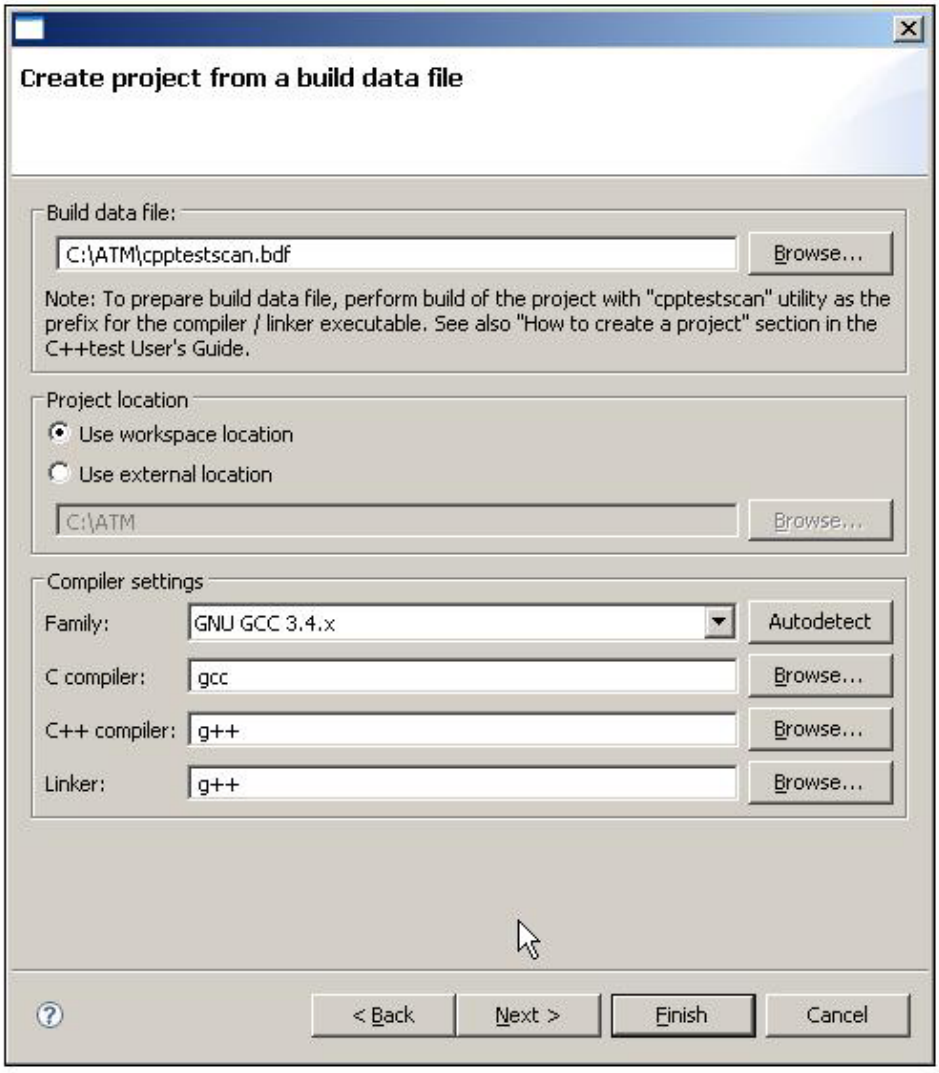
Once you have used `cpptestscan` or `cpptesttrace` to generate a build data file for code you want to test in C++test, you can use the Project Creation wizard to create a C++test project.

Custom compiler prerequisite

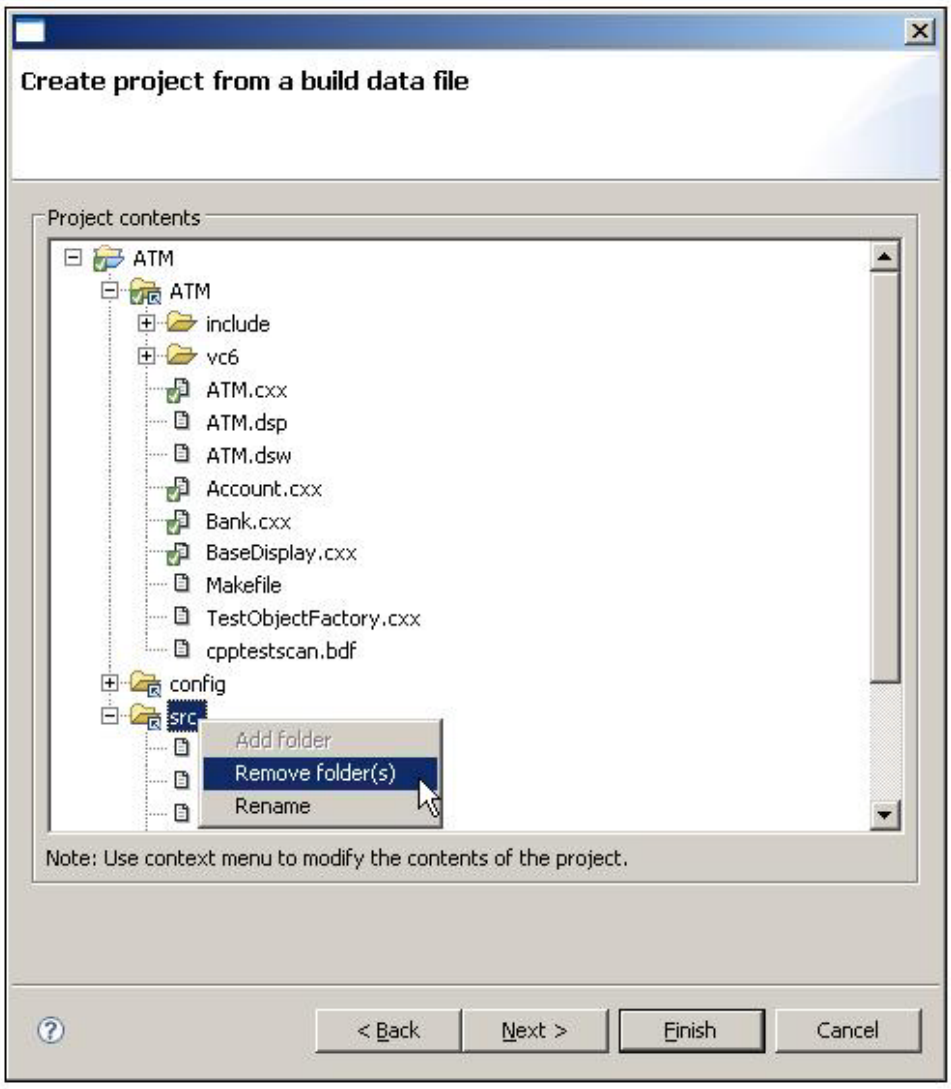
If you are using a custom compiler, add it as described in Configuring Testing with the Cross Compiler *before* starting the wizard.

To create a project from a build data file:

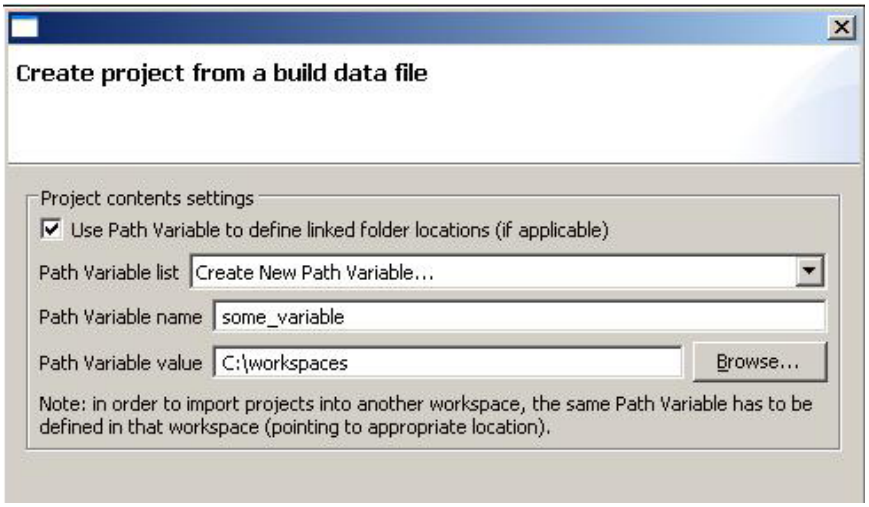
1. Open the wizard by choosing **File> New> Project**, select **C++test> Create project from a build data file**, then click **Next**. The wizard's first page will display.
2. Complete the first wizard page, then click **Next**.
    - In the **Build data file** field, enter or browse to the location of the build data file that was previously created (as described in Using cpptestscan or cpptesttrace to Create a Build Data File).
    - In the **Project location** section, specify where you want the projects created. There are two possibilities: workspace and external location. If workspace is chosen, the projects will be created in subdirectories within the workspace location. If external location is chosen, single projects will be created directly in that location. If multiple projects are created, then subdirectories for each project will be created in the specified external location.
        - For details on the available project creation options and their impacts, see Working with C++test Projects
    - In the **Compiler settings** section, specify the compiler family. The other options will be set automatically.
        - To get a list of valid compiler family values, use the `-list-compilers` switch to `cpptestcli`



3. In the second wizard page, which displays a project tree with the projects that will be created, verify the project's structure and content, making modifications as needed.
    - The first level lists all projects that will be created. To change the project's name or link additional folders to the project, right-click the project name and choose the appropriate shortcut menu command.
    - The second level lists all folders that will be linked. To change a folder`s name or prevent it from being included in the project, right-click the folder name and choose the appropriate shortcut menu command.
    - Deeper in the tree, you will see all folders and files which are present in linked folders. A green marker is used to indicate files referenced in the .bdf file.



4. (Optional) In the third wizard page, set Path Variables if needed.
   - If you want Path Variables used in linked folders, check **Use Path Variable to define linked folder locations (if applicable).**
   - To use a pre-defined Path Variable, select it from the **Path Variable list** box.
   - To use a custom Path Variable, choose **Custom** from the **Path Variable list** box, then manually enter the Path Variable name and value in corresponding fields.



Click **Finish.** C++test will create the specified project(s) in the specified location. The project(s) will include all source files whose options were scanned, and project properties should be set up appropriately.

Creating a Project from the Command Line

You can also create a BDF-based projects in command line mode by using the `-bdf <cpptestscan.bdf>` switch to `cpptestcli`.

If you want to perform analysis (e.g., static analysis and/or test generation) immediately after the project is created, ensure that the cpptestcli command uses -config to invoke the preferred Test Configuration. For example:

```
cpptestcli -data "</path/to/workspace>" -resource "<projectname>" -config "team://Team Configuration"  -localsettings "</path/to/name.properties>" -bdf  "</path/to/name.bdf>"
```

If you simply want to create the project (without performing any analysis), omit -config. For example:

```
cpptestcli -data "</path/to/workspace>" -resource "<projectname>" -localsettings "</path/to/name.properties>" -bdf  "</path/to/name.bdf>"
```

Note that -config "util/CreateProjectOnly", which was previously used for creating a project without testing, is no longer used in the current version of C++test. The fake Test Configuration "util/CreateProjectOnly" is no longer supported.

You can define custom project settings in a plain text options file, which is passed to `cpptestcli` using the `-localsettings` switch. Settings can be specified in the options file as described in Local Settings (Options) Files.

Examples

The following examples demonstrate how to create a C++test project from the command line using `cpptestscan`. They use C++test's ATM example, which is available in the `<INSTALL_DIR>/examples/ATM` directory

The examples use a make-based build; however, a .bdf file can be produced from any build system.

Assumptions and Prerequisites

- The `cpptestscan` executable should be included on $PATH (it is located in C++test's installa-tion directory).
- The `cpptestcli` executable should be included on $PATH (it is located in C++test's installa-tion directory).
- g++ is assumed to be the original compiler executable.
- The workspace and .bdf file locations have to be entered in a format supported by the given shell/command prompt. For example:
  - `/home/MyWorkspace on UNIX/Cygwin`
  - `c:\home\MyWorkspace on Windows`
  - `c:/home/MyWorkspace on Cygwin`
- To create a project without performing any testing, omit `-config`
- To create and test the new project, use the appropriate test configuration (e.g. `-config Must-HaveRules`).
- A full project rebuild will be performed ("clean all") to ensure that all objects are built in the make run

Example 1 - Creating a C++test project in the workspace location with default settings

1. Create a build data file (.bdf) based on the original Makefile as follows:
   a. Go to the `<INSTALL_DIR>/examples/ATM` directory.
   b. Build the ATM project while prefixing the original compiler executable with the `cpptestscan` executable:
      `> make CC="cpptestscan g++" clean all`
   c. Note that a new data file (`cpptestscan.bdf`) was created in the `<INSTALL_DIR>/examples/ATM` directory.
2. Create a C++test project based on the build data file (.bdf) as follows:
   a. Use C++test's CLI mode to create a new project in the `/home/MyWorkspace` workspace:
      `> cpptestcli -data /home/MyWorkspace -bdf cpptestscan.bdf`
   b. Note that a new C++test project (ATM) was created in MyWorkspace location. It con-tains all the source files and build options of the original `<INSTALL_DIR>/examples/ATM` project.

Example 2 - Creating a C++test project in original project's location with "Visual C++ 7.1" set as the compiler and "myProject" set as the project name

1. Create a build data file (.bdf) based on the original Makefile as follows:
   a. Go to the `<INSTALL_DIR>/examples/ATM` directory.
   b. Build the ATM project while prefixing the original compiler executable with the `cpptestscan` executable:
      `> make CC="cpptestscan --cpptestscanProjectName=myProject g++" clean all`
   c. Note that a new data file (`cpptestscan.bdf`) was created in the `<INSTALL_DIR>/examples/ATM` directory. Notice that `myProject` was set as project name.
2. Create a C++test project based on the build data file (.bdf) as follows:
   - First, override the default settings:
     a. Create a plain text options file named `opts.properties` in `<INSTALL_DIR>/examples/ATM`.
     b. Set the compiler family to Visual C++ 7.1 by entering `bdf.import.compiler.family=vc_7_1` into the `opts.properties` file.

c. Change the destination project location to the location of the `cpptestscan.bdf` file (which is located in the original project's directory) by entering:
`bdf.import.location=BDF_LOC into opts.properties file`

- Next, use C++test's CLI mode to create a new project in the `/home/MyWorkspace` workspace:
`cpptestcli -data /home/MyWorkspace -bdf cpptestscan.bdf -localsettings opts.properties`
- Finally, note that a New C++test project (myProject) was created in <INSTALL_DIR>/examples/ATM location, containing all the source files and build options of the original <INSTALL_DIR>/examples/ATM project, having Visual C++ 7.1 set as compiler family.

Notes:

- vc_7_1, which is listed among supported compilers, was used in this example. To use a custom compiler, you would need to specify its path in the C++test Preferences panel **(Configurations> Custom directories> Custom compilers).** See Configuring Testing with the Cross Compiler for details.
- The `BDF_LOC` variable was used as the project location; this refers to the `cpptestscan.bdf` file's location

The generated build data file can be then used to create a project from the GUI or from the command line.

Integrating C/C++test into a CMake Build

C/C++test ships with an extension for CMake that allows you to define C/C++test projects directly in the `CMakeLists.txt` build file, using the CMake syntax. The extension includes the `cpptest_add_executable()` CMake function for defining the C/C++test project, including the project location, structure, and content. As a result, the C/C++test project definition files (.project and .parasoft) and a build data file (.bdf) are automatically generated during the CMake build. When the build completes, you can import the C/C++test project files and the BDF into your workspace to perform analysis and testing.

Support for CMake integration consists of the following components:

- `<CPPTEST_INSTALL_DIR>/integration/cmake/cmake/cpptest.cmake` – the C/C++test extension for CMake you need to add to the `CMakeFiles.txt` build file to provide the C/C++test project definition.
- `<CPPTEST_INSTALL_DIR>/integration/cmake/cmake/cpptest.templates/*.in` – a set of C/C++test templates for automatically generated project definition files that allows you to highly customize the extension for CMake.

In addition, the `<CPPTEST_INSTALL_DIR>/integration/cmake` directory includes an example project to demonstrate integration with CMake using the C/C++test extension.

Requirements

- Linux 64-bit
- C/C++test Professional for Linux 64-bit
- CMake 3.10 or later

Workflow Overview

1. Include `<INSTALL_DIR>/integration/cmake/cmake/cpptest.cmake` to your `CMakeLists.txt` build file.
2. Use the `cpptest_add_executable()` function to define a target that represents your C/C++test project (see Defining the C/C++test Project for details).
3. Run CMake with the `configure` and `build` commands to generate C/C++test project configuration files.
4. Import the automatically generated C/C++test projects to your Eclipse workspace via the UI (**Import> General> Existing Projects into Workspace**) or command line (`-import <ROOT_FOLDER_OR_PROJECT_FILE>`).
5. Create test cases and configure stubs. See Test Creation and Execution for details.
6. Run unit tests.

If you store your test artifacts, such as test cases or stubs, in a source control system, do not check in the automatically generated project definition files or build data files (.bdf). These files should be generated every time your CMake project is built.

Defining the C/C++test Project

To enable CMake to automatically generate C/C++test project files, you must define a target that represents your C/C++test project with the `cpptest_add_executable()` function. At a minimum, you must must configure:

- the name of the target.
- all the source files you want to add to the C/C++test project.
- build options and dependencies (external libraries) using regular CMake functions, such as `target_include_directories()` or `target_link_libraries()`.

The following options are available:

Option Name	Description	Default
<target_name>	The name of the target.	No default. You must always configure this option.
CPPTEST_COMPIL		gcc_9-64

ER_ID	A target-specific C/C++test compiler identifier. To configure the same compiler for all your targets, specify the identifier in `cpptest.cmake`.	
CPPTEST_PROJECT_NAME	The name of the C/C++test project.	The same as the name of the target.
CPPTEST_PROJECT_LOC	The location of the C/C++test project.	The current folder.
CPPTEST_PROJECT_FOLDERS	Additional source folders you want to include in the C/C++test project. You must specify the name and location of each additional folder.	By default, the C/C++test project only includes the root directory.
EXCLUDE_FROM_ALL	If specified, the the current target is excluded form the default `all` target.	Disabled.
SOURCES	A list of sources you want to add to the C/C++test project.	No default. You must always configure this option.
TARGETS	A list of existing CMake targets. If configured, the list of source files from the specified targets are added to the C/C++test project.	No default. You must always configure this option.

```
cpptest_add_executable(
<target_name>
[CPPTEST_COMPILER_ID <compiler_id>]
[CPPTEST_PROJECT_NAME <test_project_name>]
[CPPTEST_PROJECT_LOC <test_project_location>]
[CPPTEST_PROJECT_FOLDERS <name1=location1> <name2=location2> ...]
[EXCLUDE_FROM_ALL]
SOURCES <src1.cpp> <src2.cpp> ... | TARGETS <target1> <target2> ...
)
```

Example Integration with CMake

This section demonstates integration with CMake using the example project located in the `<CPPTEST_INSTALL_DIR>/integration/cmake` directory.

1. Go to `<CPPTEST_INSTALL_DIR>/integration/cmake`.
2. If you use a compiler other than the default GNU GCC 9 (x64), replace the default compiler identifier `gcc_9-64` with your compiler identifier in `modules/mod1/CMakeLists.txt` and `cmake/cpptest.cmake`.
3. Build the example project with the following commands:

   ```
   > cd <CPPTEST_INSTALL_DIR>/integration/cmake
   > mkdir build
   > cd build
   > cmake ..
   > make
   ```

   CMake will build the project and generate the C/C++test projects. The following CMakeLists.txt files define the C/C++test projects (with different project layout and configuration options):

   - app/CMakeLists.txt
   - modules/mod1/CMakeLists.txt
   - tests/cpptest_modules/CMakeLists.txt
4. Open an empty workspace in your IDE where C/C++test is installed.
5. Choose **File> Import> General> Existing Projects into Workspace** from the IDE menu and navigate to the the `<CPPTEST_INSTALL_DIR>/integration/cmake` directory to import the three automatically generated projects to your IDE.
6. Generate test cases with the `builtin://Generate Unit Tests` test configuration.
7. Execute test cases with the `builtin://Run Unit Tests` test configuration.