GYP

# Language Specification

## Contents

## Objective

Create a tool for the Chromium project that generates native Visual Studio, Xcode and SCons and/or make build files from a platform-independent input format. Make the input format as reasonably general as possible without spending extra time trying to “get everything right,” except where not doing so would likely lead Chromium to an eventual dead end. When in doubt, do what Chromium needs and don’t worry about generalizing the solution.

## Background

Numerous other projects, both inside and outside Google, have tried to create a simple, universal cross-platform build representation that still allows sufficient per-platform flexibility to accommodate irreconcilable differences. The fact that no obvious working candidate exists that meets Chromium's requirements indicates this is probably a tougher problem than it appears at first glance. We aim to succeed by creating a tool that is highly specific to Chromium's specific use case, not to the general case of design a completely platform-independent tool for expressing any possible build.

The Mac has the most sophisticated model for application development through an IDE. Consequently, we will use the Xcode model as the starting point (the input file format must handle Chromium's use of Xcode seamlessly) and adapt the design as necessary for the other platforms.

## Overview

The overall design has the following characteristics:

- Input configurations are specified in files with the suffix `.gyp`.
- Each `.gyp` file specifies how to build the targets for the "component" defined by that file.
- Each `.gyp` file generates one or more output files appropriate to the platform:
  - On Mac, a `.gyp` file generates one Xcode .xcodeproj bundle with information about how its targets are built.
  - On Windows, a `.gyp` file generates one Visual Studio .sln file, and one Visual Studio .vcproj file per target.
  - On Linux, a `.gyp` file generates one SCons file and/or one Makefile per target
- The `.gyp` file syntax is a Python data structure.
- Use of arbitrary Python in `.gyp` files is forbidden.
  - Use of eval() with restricted globals and locals on `.gyp` file contents restricts the input to an evaluated expression, not arbitrary Python statements.
  - All input is expected to comply with JSON, with two exceptions: the # character (not inside strings) begins a comment that lasts until the end of the line, and trailing commas are permitted at the end of list and dict contents.
- Input data is a dictionary of keywords and values.
- "Invalid" keywords on any given data structure are not illegal, they're just ignored.
  - TODO: providing warnings on use of illegal keywords would help users catch typos. Figure out something nice to do with this.

## Detailed Design

Some up-front design principles/thoughts/TODOs:

- Re-use keywords consistently.
- Keywords that allow configuration of a platform-specific concept get prefixed appropriately:
  - Examples: `msvs_disabled_warnings`, `xcode_framework_dirs`
- The input syntax is declarative and data-driven.

  - This gets enforced by using Python eval() (which only evaluates an expression) instead of exec (which executes arbitrary python)

- Semantic meanings of specific keyword values get deferred until all are read and the configuration is being evaluated to spit out the appropriate file(s)
- Source file lists:
  - Are flat lists. Any imposed ordering within the .gyp file (e.g. alphabetically) is purely by convention and for developer convenience. When source files are linked or archived together, it is expected that this will occur in the order that files are listed in the .gyp file.
  - Source file lists contain no mechanism for by-hand folder configuration ( Filter tags in Visual Studio, Groups in Xcode)
  - A folder hierarchy is created automatically that mirrors the file system

## Example

```
{
  'target_defaults': {
    'defines': [
      'U_STATIC_IMPLEMENTATION',
      ['LOGFILE', 'foo.log',],
    ],
    'include_dirs': [
      '..',
    ],
  },
  'targets': [
    {
      'target_name': 'foo',
      'type': 'static_library',
      'sources': [
        'foo/src/foo.cc',
        'foo/src/foo_main.cc',
      ],
      'include_dirs': [
         'foo',
         'foo/include',
      ],
      'conditions': [
         [ 'OS==mac', { 'sources': [ 'platform_test_mac.mm' ] } ]
      ],
      'direct_dependent_settings': {
        'defines': [
          'UNIT_TEST',
        ],
        'include_dirs': [
          'foo',
          'foo/include',
          'foo/include',
        ],
      },
    },
  ],
}
```

## Structural Elements

## Top-level Dictionary

This is the single dictionary in the .gyp file that defines the targets and how they're to be built.

The following keywords are meaningful within the top-level dictionary definition:

| | |
|---|---|
| conditions | A conditional section that may contain other items that can be present in a top-level dictionary, on a conditional basis. See the "Conditionals" section below. |
| includes | A list of .gypi files to be included in the top-level dictionary. |
| target_defaults | A dictionary of default settings to be inherited by all targets in the top-level dictionary. See the "Settings keywords" section below. |
| targets | A list of target specifications. See the "targets" below. |
| variables | A dictionary containing variable definitions. Each key in this dictionary is the name of a variable, and each value must be a string value that the variable is to be set to. |

## targets

A list of dictionaries defining targets to be built by the files generated from this .gyp file.

Targets may contain includes , conditions , and variables sections as permitted in the root dictionary. The following additional keywords have structural meaning for target definitions:

| | |
|---|---|
| actions | A list of special custom actions to perform on a specific input file, or files, to produce output files. See the "Actions" section below. |
| all_dependent_settings | A dictionary of settings to be applied to all dependents of the target, transitively. This includes direct dependents and the entire set of their dependents, and so on. This section may contain anything found within a target dictionary, except configurations , target_name , and type sections. Compare direct_dependent_settings and link_settings . |
| configurations | A list of dictionaries defining build configurations for the target. See the "Configurations" section below. |

| | |
|---|---|
| copies | A list of copy actions to perform. See the “Copies” section below. |
| defines | A list of preprocesor definitions to be passed on the command line to the C/C++ compiler (via `-D` or `/D` options). |
| dependencies | A list of targets on which this target depends. Targets in other `.gyp` files are specified as `../path/to/other.gyp:target_we_want`. |
| direct_dependent_settings | A dictionary of settings to be applied to other targets that depend on this target. These settings will only be applied to direct dependents. This section may contain anything found within a `target` dictionary, except `configurations`, `target_name`, and `type` sections. Compare with `all_dependent_settings` and `link_settings`. |
| include_dirs | A list of include directories to be passed on the command line to the C/C++ compiler (via `-I` or `/I` options). |
| libraries | A list of list of libraries (and/or frameworks) on which this target depends. |
| link_settings | A dictionary of settings to be applied to targets in which this target’s contents are linked. `executable` and `shared_library` targets are linkable, so if they depend on a non-linkable target such as a `static_library`, they will adopt its `link_settings`. This section can contain anything found within a `target` dictionary, except `configurations`, `target_name`, and `type` sections. Compare `all_dependent_settings` and `direct_dependent_settings`. |
| rules | A special custom action to perform on a list of input files, to produce output files. See the “Rules” section below. |
| sources | A list of source files that are used to build this target or which should otherwise show up in the IDE for this target. In practice, we expect this list to be a union of all files necessary to build the target on all platforms, as well as other related files that aren’t actually used for building, like README files. |
| target_conditions | Like `conditions`, but evaluation is delayed until the settings have been merged into an actual target. `target_conditions` may be used to place conditionals into a `target_defaults` section but have them still depend on specific target settings. |
| target_name | The name of a target being defined. |
| type | The type of target being defined. This field currently supports `executable`, `static_library`, `shared_library`, and `none`. The `none` target type is useful when producing output which is not linked. For example, converting raw translation files into |

| | |
|---|---|
| | resources or documentation into platform specific help files. |
| msvs_props | A list of Visual Studio property sheets ( `.vsprops` files) to be used to build the target. |
| xcode_config_file | An Xcode configuration ( `.xcconfig` file) to be used to build the target. |
| xcode_framework_dirs | A list of framework directories be used to build the target. |

You can affect the way that lists/dictionaries are merged together (for example the way a list in target_defaults interacts with the same named list in the target itself) with a couple of special characters, which are covered in [Merge Basics](InputFormatReference#Merge_Basics_(=,_?,_+).md) and List Filters on the InputFormatReference page.

## configurations

`configurations` sections may be found within `targets` or `target_defaults` sections. The `configurations` section is a list of dictionaries specifying different build configurations. Because configurations are implemented as lists, it is not currently possible to override aspects of configurations that are imported into a target from a `target_defaults` section.

NOTE: It is extremely important that each target within a project define the same set of configurations. This continues to apply even when a project spans across multiple `.gyp` files.

A configuration dictionary may contain anything that can be found within a target dictionary, except for `actions` , `all_dependent_settings` , `configurations` , `dependencies` , `direct_dependent_settings` , `libraries` , `link_settings` , `sources` , `target_name` , and `type` .

Configuration dictionaries may also contain these elements:

| | |
|---|---|
| configuration_name | Required attribute. The name of the configuration. |

## Conditionals

Conditionals may appear within any dictionary in a `.gyp` file. There are two tpes of conditionals, which differ only in the timing of their processing. `conditons` sections are processed shortly after loading `.gyp` files, and `target_conditons` sections are processed after all dependencies have been computed.

A conditional section is introduced with a `conditions` or `target_conditions` dictionary keyword, and is composed of a list. Each list contains two or three elements. The first two elements, which are always required, are the conditional expression to evaluate and a dictionary containing settings to merge into the dictionary containing the `conditions` or `target_conditions` section if the expression evaluates to true. The third, optional, list element is a dictionary to merge if the expression evaluates to false.

The `eval()` of the expression string takes place in the context of global and/or local dictionaries that constructed from the `.gyp` input data, and overrides the `__builtin__` dictionary, to prevent the execution of arbitrary Python code.

## Actions

An actions section provides a list of custom build actions to perform on inputs, producing outputs. The actions section is organized as a list. Each item in the list is a dictionary having the following form:

| | | |
|---|---|---|
| action_name | string | The name of the action. Depending on how actions are implemented in the various generators, some may desire or require this property to be set to a unique name; others may ignore this property entirely. |
| inputs | list | A list of pathnames treated as inputs to the custom action. |
| outputs | list | A list of pathnames that the custom action produces. |
| action | list | A command line invocation used to produce outputs from inputs . For maximum cross-platform compatibility, invocations that require a Python interpreter should be specified with a first element "python" . This will enable generators for environments with specialized Python installations to be able to perform the action in an appropriate Python environment. |
| message | string | A message to be displayed to the user by the build system when the action is run. |

Build environments will compare inputs and outputs . If any output is missing or is outdated relative to any input , the custom action will be invoked. If all outputs are present and newer than all inputs , the outputs are considered up-to-date and the action need not be invoked.

Actions are implemented in Xcode as shell script build phases performed prior to the compilation phase. In the Visual Studio generator, actions appear files with a FileConfiguration containing a custom VCCustomBuildTool specifying the remainder of the inputs, the outputs, and the action.

Combined with variable expansions, actions can be quite powerful. Here is an example action that leverages variable expansions to minimize duplication of pathnames:

```
      'sources': [
        # libraries.cc is generated by the js2c action below.
        '<(INTERMEDIATE_DIR)/libraries.cc',
      ],
      'actions': [
        {
          'variables': {
            'core_library_files': [
              'src/runtime.js',
              'src/v8natives.js',
              'src/macros.py',
            ],
          },
          'action_name': 'js2c',
          'inputs': [
            'tools/js2c.py',
            '<@(core_library_files)',
          ],
          'outputs': [
            '<(INTERMEDIATE_DIR)/libraries.cc',
            '<(INTERMEDIATE_DIR)/libraries-empty.cc',
          ],
          'action': ['python', 'tools/js2c.py', '<@(_outputs)', 'CORE', '<@(c
        },
      ],
```

## Rules

A rules section provides custom build action to perform on inputs, producing outputs. The rules section is organized as a list. Each item in the list is a dictionary having the following form:

| | | |
|---|---|---|
| rule_name | string | The name of the rule. Depending on how Rules are implemented in the various generators, some may desire or require this property to be set to a unique name; others may ignore this property entirely. |
| extension | string | All source files of the current target with the given extension will be treated successively as inputs to the rule. |
| inputs | list | Additional dependencies of the rule. |
| outputs | list | A list of pathnames that the rule produces. Has access to RULE_INPUT_ variables (see below). |
| action | list | A command line invocation used to produce outputs from inputs . For maximum cross-platform compatibility, invocations that require a Python interpreter should be specified with a first element "python" . This will enable generators for environments with specialized Python installations to be able to perform the action in an appropriate Python environment. Has access to RULE_INPUT_ variables (see below). |
| message | string | A message to be displayed to the user by the build system when the action is run. Has access to RULE_INPUT_ variables (see below). |

There are several variables available to outputs , action , and message .

| | |
|---|---|
| RULE_INPUT_PATH | The full path to the current input. |
| RULE_INPUT_DIRNAME | The directory of the current input. |
| RULE_INPUT_NAME | The file name of the current input. |
| RULE_INPUT_ROOT | The file name of the current input without extension. |
| RULE_INPUT_EXT | The file name extension of the current input. |

Rules can be thought of as Action generators. For each source selected by extension an special action is created. This action starts out with the same inputs , outputs , action , and message as the rule. The source is added to the action's inputs . The outputs , action , and message are then handled the same but with the additional variables. If the _output variable is used in the action or message the RULE_INPUT_ variables in output will be expanded for the current source.

## Copies

A copies section provides a simple means of copying files. The copies section is organized as a list. Each item in the list is a dictionary having the following form:

| | | |
|---|---|---|
| destination | string | The directory into which the files will be copied. |
| files | list | A list of files to be copied. |

The copies will be created in destination and have the same file name as the file they are copied from. Even if the files are from multiple directories they will all be copied into the destination directory. Each destination file has an implicit build dependency on the file it is copied from.

## Generated Xcode .pbxproj Files

We derive the following things in a project.pbxproj plist file within an .xcodeproj bundle from the above input file formats as follows:

- Group hierarchy : This is generated in a fixed format with contents derived from the input files. There is no provision for the user to specify additional groups or create a custom hierarchy.
    - Configuration group : This will be used with the xcode_config_file property above, if needed.
    - Source group : The union of the sources lists of all targets after applying appropriate conditions . The resulting list is sorted and put into a group hierarchy that matches the layout of the directory tree on disk, with a root of // (the top of the hierarchy).
    - Frameworks group : Taken directly from libraries value for the target, after applying appropriate conditions.
    - Projects group : References to other .xcodeproj bundles that are needed by the .xcodeproj in which the group is contained.
    - Products group : Output from the various targets.
- Project References :
- Project Configurations :
    - Per- .xcodeproj file settings are not supported, all settings are applied at the target level.
- Targets :
    - Phases : Copy sources, link with libraries/frameworks, ...
    - Target Configurations : Specified by input.

    - Dependencies : (local and remote)

## Generated Visual Studio .vcproj Files

We derive the following sections in a `.vcproj` file from the above input file formats as follows:

- VisualStudioProject :
    - Platforms :
    - ToolFiles :
    - Configurations :
    - Configuration :
    - References :
    - Files :
    - Filter :
    - File :
        - FileConfiguration :
        - Tool :
    - Globals :

## Generated Visual Studio .sln Files

We derive the following sections in a `.sln` file from the above input file formats as follows:

- Projects :
    - WebsiteProperties :
    - ProjectDependencies :
- Global :
    - SolutionConfigurationPlatforms :
    - ProjectConfigurationPlatforms :
    - SolutionProperties :
    - NestedProjects :

# Caveats

Notes/Question from very first prototype draft of the language. Make sure these issues are addressed somewhere before deleting.

- Libraries are easy, application abstraction is harder
    - Applications involves resource compilation
    - Applications involve many inputs
    - Applications include transitive closure of dependencies
- Specific use cases like cc_library
    - Mac compiles more than just .c/.cpp files (specifically, .m and .mm files)

  - Compiler options vary by:
  - File type
  - Target type
  - Individual file
  - Files may have custom settings per file per platform, but we probably don’t care or need to support this in gyp.

- Will all linked non-Chromium projects always use the same versions of every subsystem?
- Variants are difficult. We’ve identified the following variants (some specific to Chromium, some typical of other projects in the same ballpark):

  - Target platform
  - V8 vs. JSC
  - Debug vs. Release
  - Toolchain (VS version, gcc, version)
  - Host platform
  - L10N
  - Vendor
  - Purify / Valgrind

- Will everyone upgrade VS at once?
- What does a dylib dependency mean?

Powered by Gitiles