BuSyB User's Guide

Developing and maintaining the build process of complex software systems with BuSyB.

Texas Instruments Inc.
Carsten Krueger <ckr@ti.com>
Edited by Joachim RichterErwin Schmid

BuSyB User's Guide: Developing and maintaining the build process of complex software systems with BuSyB.

by Texas Instruments Inc., Carsten Krueger, Joachim Richter, and Erwin Schmid

Published May 13, 2004
Copyright © 2002, 2003, 2004 Texas Instruments, Inc. All rights reserved.

Project	TCS 3.0
ID	89_03_15_00479
Version	1.0.1
Title	BuSyB User's Guide
Status	Being Processed

Every effort has been made to ensure that the information contained in this document is accurate at the time of printing. However, the software described in this document is subject to continuous development and improvement. Texas Instruments reserves the right to change the specification of the software. Information in this document is subject to change without notice and does not represent a commitment on the part of Texas Instruments. Texas Instruments accepts no liability for any loss or damage arising from the use of any information contained in this document.

The software described in this document is furnished under a license agreement and may be used or copied only in accordance with the terms of the agreement. It is an offence to copy the software in any way except as specifically set out in the agreement. No part of this document may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying and recording, for any purpose without the express written permission of Texas Instruments.

Table of Contents

List of Figures

Chapter 1. Introduction

This document describes the tool BuSyB and its underlying concepts. It should put the reader in a position to use BuSyB for the definition of the build process of any software system. All information in this document refers to version 1.0.4 of the BuSyB software.

The chapter following this introduction will explain the basic concepts that directed the development of BuSyB. The underlying concepts include the classification of all information that is part of a build process and the mapping of these classes to certain XML document types. Also the cooperation of BuSyB and Make for the actual build and the position of the two tools in the software lifecycle will be highlighted. The next chapter is dedicated to the usage of BuSyB. Each aspect of a build system (i.e. each class of information) is covered in a separate section. While advancing through these sections the build process for a simple example is developed. At the end of this chapter the build process of the example has been completely defined.

Today's software systems are often very complex. They are built out of great many files using different tools. Usually they come in several variants, e.g. for different operating systems or supporting different feature sets. Modern software paradigms using high-level description languages for both documentation and implementation and their extensive use of code generators additionally contribute to the complexity of the build process. The distribution of the software development between various persons, offices and even sites is another challenge to be met by a build system.

Tradditional Make based build systems fail to fulfill these new requirements. The complexity of the software system and the great number of developers working on it lead to complex, hard-to-understand makefiles and build systems that are difficult to maintain.

BuSyB is a tool for developing and maintaining the build process for complex software products. It is based on a clear classification of all information that is part of the build process, e.g. information about the tools, information about the variants. Given this classification BuSyB allows a redundancy reduced specification of the build process. This specification is stored in a number of XML files. These files are human readable and thus can be “diff'ed” and “merged”. From this set of XML files BuSyB generates a makefile, which can then be processed by Make. The generated makefile does not use any sophisticated feature of a specific Make, i.e. compatibility with Clearmake and GNU Make has been tested.

With the application of this “Best Fit Technology” - BuSyB for information entry and validation, XML for information storage and Make for the execution of the build - it is easy to create build systems that are despite of their complexity robust and easy to maintain. Each of the three “technologies” is used in its original area of application and thus “best fits” its purpose.

1.1. Prerequisites

BuSyB has been developed as a Java™ application. The XML support was achieved through the Apache XERCES XML parser and the XOM XML object model. These software packages are not part of BuSyB and must be supplied by the user.

BuSyB has been tested with the following software versions:

- Java™ - version 1.4.2
- XERCES - version 2.5.0

- XOM - version 1.0d22

Chapter 2. Concepts

2.1. Build Information - Classification

The information being part of a build system is manifold. It can be classified by its impact on the build, i.e. by the part of the build process that it influences. For each of these classes (or aspects) one document type exists (except for the Main Properties, which are spread over two document types). The following information classes can be identified, the names of the associated document types are noted in parentheses:

- Main Properties (`ConfigSet` , `ConfigDef`) - this class describes valid variants and supported feature sets.
- Target Properties (`TargetSet`) - this class contains information about what is built (e.g. executables, libraries, object files, etc.).
- Common Options (`OptionSet`) - this class covers options that are common to every software module.
- Module Properties (`SourceSet`) - this class covers the information that is specific to a software module.
- Tool Properties (`ToolSet`) - this class contains information about the tool (e.g. their command line specifics).
- Path Properties (`PathSet`) - this class describes where the files, tools etc. are found in the filesystem.

Each document type and the information which it can contain is explained in detail in Chapter Usage.

2.1.1. Main Properties

Main Properties are items that influence the software to be built in a very general manner. Usually only a few such properties exist. They are characterized by the fact that they have an impact on almost every module. They may influence also which tools are chosen and how these tools are used during the build. Examples for such Main Properties are:

- the operating system for which the software should be build,
- the feature set that is supported by the software,
- whether the software contains debug information or not.

Main Properties can have certain relations to each other[1]. This relation can either be exclusion or implication or none. None means that two properties are completely independent from each other. Their presence can be chosen individually. Exclusion means that the presence of one such property excludes the presence of the other property. And implication means that one property can only be present if the other is present too.

[1]This feature has not been fully implemented yet.

An example for such a relation can be taken from a GSM/GPRS protocol stack. One feature one can choose is whether the software contains a WAP browser or not. Another property is whether it contains a TCP/IP module. It makes sense to have a software that has TCP/IP support but no WAP Browser, but it does not make sense have WAP without TCP/IP. Thus WAP implies (or depends on) TCP/IP.

2.1.2. Target Properties

Target Properties are those items that describe the output of the build process or - using Make vocabulary - its targets. Targets are files (or directories) created from other files [2] using a specified tool. Targets can also be composed of other (sub)targets. BuSyB allows to specify sets of targets. For the generation of the makefile one target has to be chosen. The makefile will only contain the rules necessary to build exactly this target.

2.1.3. Common Options

Common Options are items that are passed to the tools during the build process. They are common for all (or at least for many) modules. Options specific to just one module are covered in another category (Module Properties). Examples for Common Options are:

- global include directories passed to the C-Compiler,
- global preprocessor definitions passed to the C-Compiler,
- common flags passed to a certain tool.

2.1.4. Module Properties

Module Properties are all pieces of information that are specific to a single software module. This includes the source files, specific options and other settings. A software module is understood as a closely coupled set of files which share the same options, the same file extension and which are combined (possibly with more files) into a single target.

2.1.5. Tool Properties

Tool Properties are all those properties that are related to the tools used during the build process, i.e. compiler, linker, scripts, genererators. These properties include among other things the specifics of the command line, the parameters and the calling modell (executable vs. script).

2.1.6. Path Properties

The information where files or folders are located is central to the build process. Source files, output files, tools - for all these the information where they can be found or placed in the file system is necessary. In addition to that most of this path information is not only needed in one place but in many places. Centralisation of this information allows to enter it without redundancy. Examples for Path Properties are:

- the directory where the source files of a module are located,

[2] In future version of the tool it will also be possible to specify targets that are generated without explicit input files, e.g. Make rules without prerequisites.

- the location of an include directory,
- a directory where intermediate files go.

A well known practice is to have rules for the naming of certain files or directories, e.g. the name (final) software product, follows a naming scheme expressing the feature set the software contains and the platform it runs on (xyz_full_win32.exe). The information that the name of a file or directory is created following a certain rule can be part of its Path Properties.

2.2. Build System - Best Fit Technology

Designing a build system with BuSyB is based on three core technologies. On the tool BuSyB itself for the definition of the build system, on XML for the information storage and on Make for the actual build process. From these three technologies a foundation is formed by combining the individual strengths of each of them.

2.2.1. BuSyB

BuSyB allows to produce a precise and concise specification of the build process. Given the clear classification of build relevant information, BuSyB supports the definition of a build process with context sensitive editing and a validation of the build system. BuSyB generates makefiles which can be processed by standard Make.

The big advantage of using BuSyB is that it is especially designed to describe the build process of complex software systems. It focuses on a clearly structured description of the build process and helps developing it.

BuSyB is developed using Java. The Java runtime environment is available for all major platforms thus also BuSyB is nearly platform independent.

2.2.2. XML

All information needed to describe the build process is stored in XML files. These files are based on a DTD developed especially for this purpose. Seven distinct document types have been defined within the DTD. Each of these document types is suitable for the specification of a certain class of information about the build process. These document types and the aspect of the build process which they define are:

- `ConfigSet` - Definition of Main Properties (all possible variants),
- `ConfigDef` - Selection of Main Properties (the actual variants),
- `TargetSet` - Definition of Target Properties,
- `OptionSet` - Definition of Common Options,
- `SourceSet` - Definition of Module Properties,
- `ToolSet` - Definition of Tool Properties,
- `PathSet` - Definition of Path Properties.

XML with its pure textual representation is ideal for information storage. The XML files can be subject to comparison and merging. Not only BuSyB but also ordinary XML editors can be used to manipulate them. Although context sensitive help is available only to a lesser extend then. Another big advantage of using XML here is that because of its nature a special parser/lexer is not necessary but standard frameworks can be used for this.

2.2.3. Make

Make is widely accepted as the best tool for building (and rebuilding) software. Given a makefile Make automatically determines which pieces of a software system need to be recompiled and issues the commands to recompile them. Make is fast, mature, reliable and available on almost every platform.

2.3. Build Process - Lifecycle

As the the definition of a build system is based on three core technologies (BuSyB,XML,Make), the actual build process is not only one but a series of activities. First of all the complete build system must be specified with BuSyB. The result is a set of XML documents. Each set of files describes exactly one build system. This does not mean that only one target (one piece of software that is produced) can be build. Instead many targets are part of the build system.

From a set of XML documents a makefile must be generated as a second step. This makefile contains rules for every target that has been defined.

This makefile can now be processed by Make. Make builds either every target or if explicitly given only chosen targets.

If changes to the build system are necessary they must be incorporated using BuSyB or an ordinary XML editor. Upon that a new makefile must be generated with BuSyB. This new makefile can then again be processed by Make

The generated makefiles are such that they can be distributed as they are. No other tools apart from Make and those tools needed during the build are necessary to actually perform the build. Of course these makefiles can be modified - in fact their structure is very simple, one rule for every target, almost no variables - but normally this should not be necessary.

Chapter 3. Using BuSyB

Creating a simple example

This chapter demonstrates the usage of BuSyB. BuSyB is heavily based on the underlying DTD. Not only the content of the resulting XML files but also the name of GUI elements are based on it. Therefore the DTD has a central place in the remainder of this chapter. Each of the following sections covers one class of build information and thus one document type. The structure of each section is as follows.

The first part of the section describes the part of the DTD that is defining the document type relating to this information class. The elements of the DTD and their meaning will be explained in detail.

The second part is a simple example. This example is first described verbally followed by a specification in XML. Critical places or spots that require special attention will be explained.

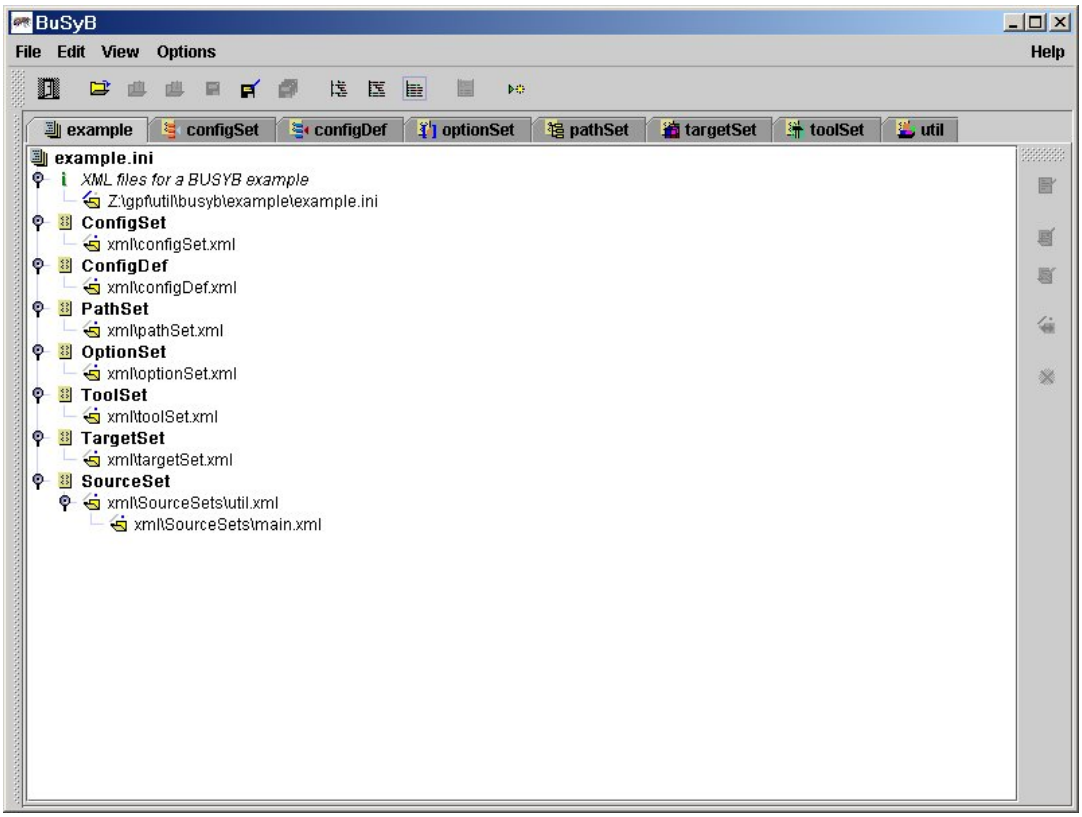
An optional third part shows a snipped of the generated makefile that corresponds to the example document described in the previous section. It will be show which information goes where in the makefile. Please note that this section is only present when applicable.

The fourth part is dedicated to the tool BuSyB. It will be explained how BuSyB is used and how the example is handled by the tool.

3.1. General

BuSyB presents the content of the files in a tree oriented fashion (vs. a tabular oriented style of some other XML editors). When initially loading a file, its structure is not fully expanded. Instead the user can navigate to the place of interest by expanding the elements and attributes.

Figure 3.1. General appearance of BUSYB



3.2. The Example

The example has been designed to illustrate the main aspects of defining a build system with BuSyB while staying as simple as possible. The application whose build process is specified does not make to much sense but it serves only educational purposes.

The application, whose build process is defined, is a Windows™ program. It will be build using Microsoft™ tools.

The programm is composed of 3 C-files - one header file (`util.h`) and two source files (`main.c` and `util.c`).

The application can be built in two main variants “debug” and “non debug”. The “non debug” variant is composed only of one object `main.c`. The “debug” variant contains additionally the functionality implemented in a library `util.lib` created from the object `util.obj`.

All input files and all generated files are distributed in a certain directory structure (separating all output of the build process for the “debug” variant from those of the “non debug” variant.

The tools (i.e. compiler and linker) used to compile the program are found through the environment variable PATH. Part of the build process is automatic dependency generation using two more tools (CPP and a Perl script). The location of the script is explicitly defined using a relative path.

3.3. Main Properties

The `ConfigSet` and `ConfigDef` Documents

The `ConfigSet` and `ConfigDef` document types are designed for the specification of the Main Properties of the build system. Usually only a few such properties exist but they impact the build process almost everywhere. The definition of these properties has been split into two document types.

The `ConfigSet` document type defines the properties of the build system by giving each property a specific name. For each property a list of allowed values is specified, too. [3]

The `ConfigDef` document chooses one set of properties by specifying a value for each property that is of interest.

ConfigSet and ConfigDef

The `ConfigSet` document describes all possible combinations of properties. It must therefore be defined by someone knowing all Main Properties and their relations to each other. This document should be carefully designed such that only the selection of valid combinations is possible. [4]

The `ConfigDef` document chooses exactly one of these combinations. Assuming that the `ConfigSet` document has been carefully designed only valid versions can be chosen.

3.3.1. DTD for `ConfigSet` Documents

Figure 3.2. DTD for `ConfigSet` Documents

```
<!ELEMENT configSet     ( comment*, propertyGroup*, propertySet* )>    (1)
<!ATTLIST configSet       name        CDATA #REQUIRED
                          description CDATA #REQUIRED>
<!ELEMENT propertyGroup ( comment*, propertySet* )>                    (2)
<!ATTLIST propertyGroup   name        CDATA #REQUIRED
                          description CDATA #REQUIRED>
<!ELEMENT propertySet   ( comment*, valueDef+ )>                       (3)
<!ATTLIST propertySet     name        CDATA #REQUIRED
                          description CDATA #REQUIRED
                          shortName   CDATA #REQUIRED>
<!ELEMENT valueDef      ( comment* )>                                  (4)
<!ATTLIST valueDef        value       CDATA #REQUIRED
                          description CDATA #REQUIRED
                          valname     CDATA #IMPLIED>
```

1. The `ConfigSet` element contains a list `propertySet`s. It can be attributed with a *name* and a *description*.
2. Groups of properties can be constructed with the `propertyGroup` element. These groups can be referenced from other document types.
3. Each `propertySet` element defines one property. The *name* attribute specifies a symbolic name through which this property can be referenced from all other document types, i.e. many elements have a *require* attribute in which conditions of the form NAME==value can be specified. The *shortName* attribute may contain a string that can be used for the automatic generation of (files, directory, ...) names.
4. Each `propertySet` element contains a list of `valueDef`s. Each `valueDef` defines a legal value for this property. The *value* attribute takes in this value which must be an integer. The *description* attribute can contains a verbal description of this value. The *valname* attribute may contain a string that can be used for the automatic

[3] In future releases relations between properties (exclusion, implication, ...) may also be specified here.
[4] Relations between properties are not yet implemented in BuSyB

generation of names for files or directories.

3.3.2. Sample `ConfigSet` Document

This example defines one Main Property (`DEBUG_MODE`) which has two legal values. This property will be referenced throughout the the example.

Figure 3.3. Sample `ConfigSet` Document

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE configSet SYSTEM "bsbDocs.dtd">
<configSet description="Example - Main Properties" name="properties">
  <propertySet description="define kind of debug" name="DEBUG_MODE"    (1)
               shortName="">
    <valueDef description="No Debug" value="0" valname=""/>
    <valueDef description="Debug with help functions" value="1"
    valname="_db"/>                                                    (2)
  </propertySet>
</configSet>
```

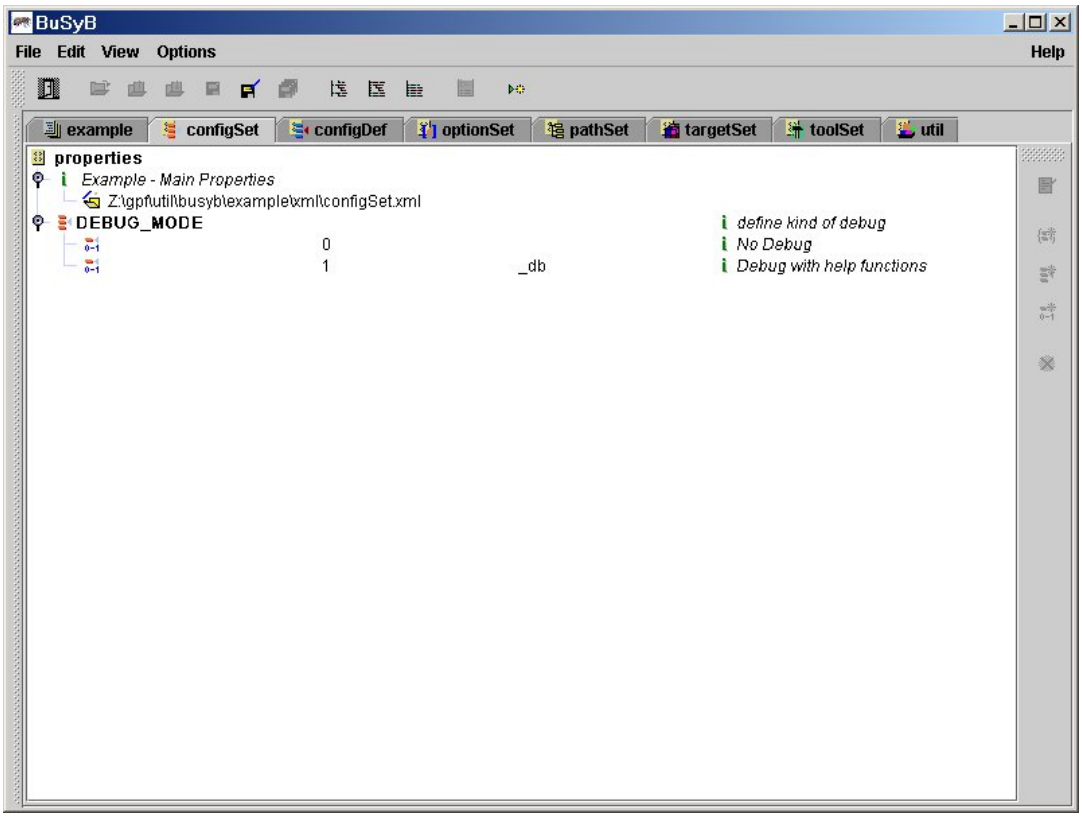
1 Here the property `DEBUG_MODE` is defined, which takes two legal values `0` for the “non debug” and `1` for the “debug” version.

2 The attribute *valname* contains a string that will later be used for the construction of the name of certain directories.

3.3.3. Sample `ConfigSet` editing with BuSyB

The following picture shows how the contents of the previously defined `ConfigSet` document are displayed by BuSyB.

Figure 3.4. Sample `ConfigSet` editing with BuSyB



3.3.4. DTD for `ConfigDef` Documents

Figure 3.5. DTD for `ConfigDef` Documents

```
<!ELEMENT configDef     ( comment*, property* )>                    (1)
<!ATTLIST configDef       name        CDATA #REQUIRED
                          description CDATA #REQUIRED
                          reference   CDATA #IMPLIED>
<!ELEMENT property      ( comment* )>                               (2)
<!ATTLIST property        name        CDATA #IMPLIED
                          value       CDATA #IMPLIED
                          include     CDATA #IMPLIED>               (3)
```

1 The `ConfigDef` element contains a list of `property` elements. It must be attributed with a *name* and a *description*. The *reference* attribute may contain the filename of an associated BuSyB ".ini" file. If BuSyB is invoked with option `-cfg`, this attribute determines the set of documents (`ConfigSet`, `PathSet`, `TargetSet`, etc.) to be used for this specific configuration. The reference attribute (if present at all) may be empty or equivalently contain the string "none" (for historical reasons;-); in these cases BuSyB requires a “standard” "ini"-file.

2 Each `property` element defines the value of one Main Property of the system. The attribute *name* references one defined `propertySet` element of the associated `ConfigSet` document. The *value* attribute specifies one legal value for this property.

3 The attribute *include* can be used to include the contents of other `ConfigDef` documents. Therefore the name of the document that should be included must be specified in the *include* attribute. `property` elements with the same name are allowed in different `ConfigDef` documents; the last one (in a textual sense) will be evaluated by

BuSyB.

3.3.5. Sample `ConfigDef` Document

This example defines that out of two possible configurations the “debug” variant is build.

Figure 3.6. Sample `ConfigDef` Document

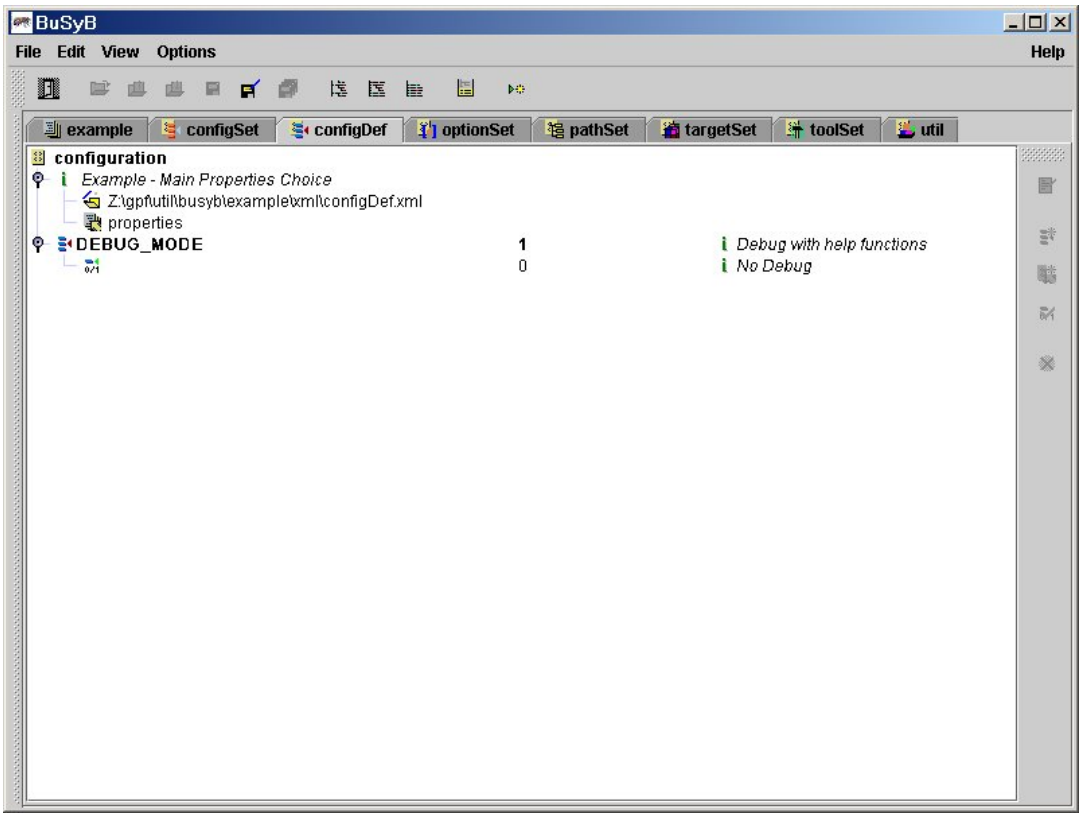
```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE configDef SYSTEM "bsbDocs.dtd">
<configDef description="Example - Main Properties Choice"
  name="configuration" reference="properties">
  <property name="DEBUG_MODE" value="1"/>                                (1)
</configDef>
```

1 For the `property DEBUG_MODE` the value `1` (“debug”) is chosen i.e using this `ConfigDef` document the “debug” variant will be build.

3.3.6. Sample `ConfigDef` editing with BuSyB

The following picture shows how the contents of the previously defined `ConfigDef` document are displayed by BuSyB. Please note that BuSyB displays also some information that is not found in this document but instead found in the `ConfigSet` document. In addition to currently choosen values (“debug” as defined in `ConfigDef` document) all other possible (“non debug”) values are shown and can be selected.

Figure 3.7. Sample `ConfigDef` editing with BuSyB



3.4. Target Properties

The `TargetSet` Document

The `TargetSet` document contains a set of targets, i.e. things to be produced during the build. Each `target` can be composed of (sub)`target`s. Thus a tree like structure representing the composition of the "root" `target` can be formed. It is possible to form more than one of these "build trees" within one `TargetSet` document. Thus different software builds can be described within one `TargetSet` document.

For the makefile generation exactly one "root" `target` must be chosen. If not, the first `target` found in the `TargetSet` will be chosen by default. Only the "build tree" defined by this `target` will be transformed into the makefile, e.g. the generator must be called twice - with a different `target` each time - if two "build trees", i.e. two independent `target`s, are described in the `TargetSet` document. The BuSyB command line is defined in the command line section of this document.

3.4.1. DTD for `TargetSet` Documents

Figure 3.8. DTD for `TargetSet` Documents

```
<!ELEMENT targetSet     ( comment*, targetInc*, target+ )>            (1)
<!ATTLIST targetSet       name        CDATA #REQUIRED
                          description CDATA #REQUIRED>
<!ELEMENT targetInc     ( comment* )>                                 (2)
<!ATTLIST targetInc       require     CDATA #IMPLIED
                          includePath CDATA #IMPLIED
                          includeFile CDATA #REQUIRED
                          description CDATA #REQUIRED>
```

```
<!ELEMENT target        ( comment*, settings, targetRef*, targetDef* )>(3)
<!ATTLIST target          require     CDATA #IMPLIED                    (4)
                          name        CDATA #REQUIRED
                          autoNamed   CDATA #IMPLIED
                          description CDATA #REQUIRED>
<!ELEMENT targetRef     ( comment* )>                                   (5)
<!ATTLIST targetRef       require     CDATA #IMPLIED
                          target      CDATA #REQUIRED>
<!ELEMENT targetDef     ( comment*, targetPath+, sources* )>            (6)
<!ATTLIST targetDef       require     CDATA #IMPLIED
                          tool        CDATA #IMPLIED
                          localFlags  CDATA #IMPLIED
                          type        CDATA #IMPLIED>
<!ELEMENT targetPath    ( comment* )>
<!ATTLIST targetPath      require     CDATA #IMPLIED                    (7)
                          pathRef     CDATA #IMPLIED
                          path        CDATA #IMPLIED>
<!ELEMENT sources       ( comment*, sourceFile*, targetDef? )>          (8)
<!ATTLIST sources         require     CDATA #IMPLIED
                          type        CDATA #IMPLIED
                          sourceSet   CDATA #IMPLIED
                          target      CDATA #IMPLIED
                          localFlags  CDATA #IMPLIED>
<!ELEMENT sourceFile    ( comment* )>
<!ATTLIST sourceFile      require     CDATA #IMPLIED
                          pathRef     CDATA #IMPLIED
                          path        CDATA #IMPLIED
                          localFlags  CDATA #IMPLIED>
```

1 The `TargetSet` element contains a list of `target`s and optional a list of include specifications. It can be attributed with a *name* and a *description*.

2 If a `targetInc` element is given, the specified file - which must be a valid `TargetSet` document - will be included as a set of targets. The file specification consists of two attributes; the (optional) *includePath* attribute contains a reference to a node in the current `PathSet`, while the *includeFile* attribute specifies the `TargetSet` document to be included. The *includeFile* may be given as an absolute or relative path. If *includeFile* is absolute, the *includePath* attribute will be ignored by BuSyB. If *includeFile* is relative, it will be evaluated relative to the specified node in the current `PathSet` (if *includePath* is present) or to the location of the including `TargetSet` document. The *description* attribute should contain a note on the included `TargetSet`. The *require* attribute controls whether the specified file will be included dependent on the current configuration settings.

3 Each `target` element defines a “makeable” target, i.e. a file that is to be produced during the build. The `target` contains (a possibly empty) `settings` element. This can be used to define `target` specific `options` as defined in the `OptionSet`.

4 A `target` can be attributed by a *description*. Two additional attributes define the name of the target - *name* and *autoNamed*. This name of a target is used for two purposes. First it is used for references within the `TargetSet` document, e.g. to construct a `target` from (sub)`target`s. Second it will become part of the name of the actually produced file during the build, e.g. the name of a library. The *autoNamed* attribute can be used to reference a name construction rule defined in the `OptionSet` document. The *require* attribute allows for multiple `target`s with the same name (buddies). Exactly one of the “buddy”-requirements must evaluate to *true* with respect to the current configuration, otherwise an error message is generated, and the resulting makefile will probably be incorrect.

5 A `target` may contain a list of `targetRef` elements. Each `targetRef` element references a `target` that must become part of the makefile even if it is not part of the actual “build tree” for that `target`, e.g. the documentation of a software may included into the makefile by a `targetRef` even though it should not become part of the actual executable file.

6 Each `target` contains a list of `targetDef` elements. A `targetDef` element defines how a `target` is built. Though more (and less) than one `targetDef` ele-

ment is allowed here, exactly one must be valid at a given time. That means in case of multiple `targetDef` elements each of them must be restricted by mutually exclusive conditions in the *require* attribute. The *tool* attribute references the `tool` used to build the target. The *type* attribute may contain a string (file extension) that will together with the `TargetSet`s name form the complete name of the file. The *localFlags* attribute may contain special arguments, which need to be passed to the `tool` as defined in the `ToolSet` document.

7 The `targetDef` element contains a list of `targetPath` elements. The `targetPath` defines the directory where the output file should be produced. This path can be constructed with the *path* and *pathRef* attributes analogous to the procedure described in the `OptionSet` document. Though more than one `targetPath` element is allowed here, exactly one must be valid at a given time. That means in case of multiple `targetPath` elements each of them must be restricted by mutually exclusive conditions in the *require* attribute.

8 The `targetDef` element contains a list of `sources` elements. Each `sources` element describes one contributor to the `target`, i.e. all of them form the list of files from which the `target` is built. Four distinct possibilities exist to define such a `sources` element. The element `sourceFile` allows to specify directly a (physically present) file, e.g. a third party library that must not be built. The attribute *target* allows to reference a (sub)`target` defined in the `TargetSet` document. The attribute *sourceSet* allows to reference a `SourceSet` defined in the `SourceSet` document, i.e. all files defined in the `SourceSet` will become part of the `target`. The fourth possibility using the element `targetDef` allows the definition (instead of a reference with the attribute *target*) of a (sub) `target` inline, e.g. for a library that is composed by the archiver from the object files that a compiler produced from all files of a `SourceSet` document.

3.4.2. Sample `TargetSet` Document

This example defines the target (`hello`). This target is built of two (sub)targets (`main` and `util`). The resulting makefile will contain rules for `hello.exe`, `main.obj` and `util.lib`.

Figure 3.9. Sample `TargetSet` Document

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE targetSet SYSTEM "bsbDocs.dtd">
<targetSet description="Example - Targets Properties" name="targets">
  <target description="executable Image" name="hello">                    (1)
    <settings/>
    <targetDef tool="LNK" type="exe">                                     (2)
      <targetPath pathRef="TARGET_OUT"/>                                  (3)
      <sources target="main"/>
      <sources target="util" require="DEBUG_MODE==1"/>                    (4)
    </targetDef>
  </target>
  <target description="main objects" name="main">
    <settings/>
    <targetDef tool="CC" type="obj">                                      (5)
      <targetPath pathRef="OUT_LIB"/>
      <sources sourceSet="MAIN"/>
    </targetDef>
  </target>
  <target description="util library" name="util">
    <settings/>
    <targetDef tool="AR" type="lib">                                      (6)
      <targetPath pathRef="OUT_LIB"/>
      <sources>
        <targetDef tool="CC" type="obj">
          <targetPath pathRef="OUT_OBJ_UTIL"/>
```

```
          <sources sourceSet="UTIL"/>
        </targetDef>
      </sources>
    </targetDef>
  </target>
</targetSet>
```

1 Here the target `hello` is defined.
The target is built using the tool `LNK` defined in the `ToolSet` document. The file name (`hello.exe`) is constructed from the name of the target and the type.

3 The resulting file (`hello.exe`) will be placed in the directory `TARGET_OUT` defined in the `PathSet` document.

4 The target `hello` is combined of two (sub)targets `main` and `util`. The target `util` will only be part of `hello` (and also only than appear in the makefile at all) if the condition `DEBUG_MODE==1` is true.

5 The target `main` is built from all the files in the `SourceSet main` as defined in a `SourceSet` document. The output will be placed in the directory `OUT_LIB` defined in the `PathSet` document. The resulting files produced by the tool `CC` will get the file extension `.obj`.

6 The target `util` is built in two steps. This is done defining an (anonymous) target inside `util`. The anonymous target comprises the results of the tool `CC` applied to all files from the `UTIL SourceSet` document. The target `util` is then composed by the tool `AR`. The resulting file will be named `util.lib` and will be placed in the `OUT_OBJ_UTIL` directory.

3.4.3. `TargetSet` information in the makefile

The following makefile snippet illustrates how various parts of the previously defined example `TargetSet` document are represented in the makefile. Please note, that this snipped may not be found exacly like this the makefile. The relative order is preserved but some lines may have been stripped out for readability.

Figure 3.10. `TargetSet` information in the makefile

```
BuSyB-DefaultTarget: hello                                                    (1)

out/out_db/hello.exe:  \                                                      (2)
        out/out_db/lib/main.obj \                                             (3)
        out/out_db/lib/util.lib
        link /debug /PDB:NONE /incremental:no /subsystem:console \            (4)
        /OUT:out/out_db/hello.exe \
        $^

hello: \
        out/out_db/hello.exe

clean_hello:                                                                  (5)
        $(BSB_REMOVE) out/out_db/hello.exe

clean:  \
        clean_main \
        clean_util \
        clean_hello
```

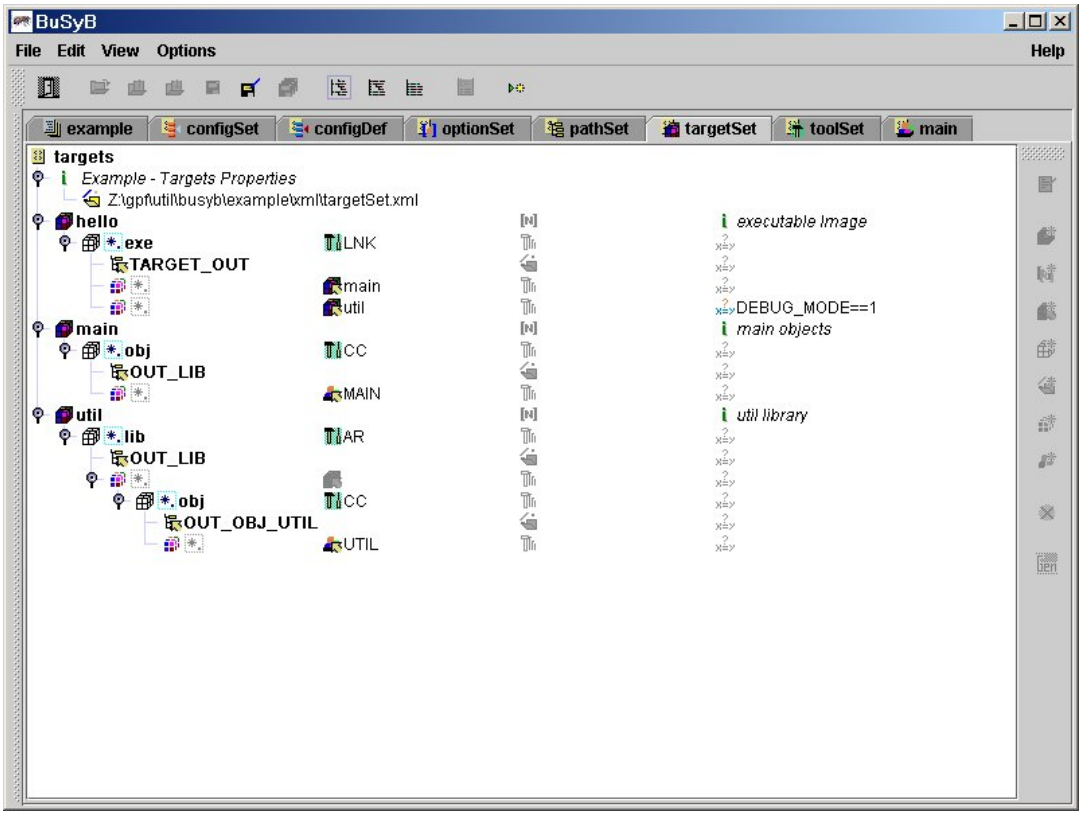
1 This rule can be found at the beginning of the makefile. It is the first rule and thus makes the target `hello` the default target.

2 The real file behind `hello` is `hello.exe` in the directory `out/out_db`.
3 The target `hello` is build from the files `main.obj` and `util.lib`, which are the results of the building (sub)targets `main` and `util`.
4 The file `hello.exe` is produced by the tool `link`
5 BuSyB automatically inserts clean rules.

3.4.4. Sample `TargetSet` editing with BuSyB

The following picture shows how the contents of the previously defined `TargetSet` document are displayed by BuSyB.

Figure 3.11. Sample `TargetSet` editing with BuSyB



3.5. Common Options

The `OptionSet` Document

The `OptionSet` document type is designed to contain all those options that are used for many modules. These options are items passed to tools. They may appear in the directly tool's command line either as they are or prefixed or otherwise modified. Options may be conditionally set depending on certain Main Properties.

3.5.1. DTD for `OptionSet` Documents

Figure 3.12. DTD for `OptionSet` Documents

```
<!ELEMENT optionSet     ( comment*, options )>                          (1)
<!ATTLIST optionSet       name        CDATA #REQUIRED
                          description CDATA #REQUIRED>
<!ELEMENT options       ( comment*, optionDef+ )>
<!ELEMENT optionDef     ( comment*, condOption+ )>                      (2)
<!ATTLIST optionDef       name        CDATA #IMPLIED
                          description CDATA #REQUIRED>
<!ELEMENT condOption    ( comment*, condValue* )>                       (3)
<!ATTLIST condOption      require     CDATA #IMPLIED
                          name        CDATA #IMPLIED
                          optRef      CDATA #IMPLIED>
<!ELEMENT condValue     ( comment* )>                                   (4)
<!ATTLIST condValue       require     CDATA #IMPLIED
                          pathRef     CDATA #IMPLIED                  (5)
                          path        CDATA #IMPLIED
                          grpRef      CDATA #IMPLIED                  (6)
                          valRef      CDATA #IMPLIED                  (7)
                          value       CDATA #IMPLIED>                 (8)
<!ELEMENT settings      ( comment*, options? )>
```

1 The `OptionSet` element contains a list of `options`. It can be attributed with a *name* and a *description*.

2 Each `options` element contains a list of `optionDef`s. An `optionDef` is the definition of a group of options. If an `optionDef` is given a *name* attribute it can be referenced using this *name* from other document types. An `optionDef` is designed to contain all those options that are referenced and used together, e.g. all options that need to be passed to the pre-processor (defines and includes).

3 Each `optionDef` contains a list `condOption`s. An `condOption` is the definition of a more specific group of options. In contrary to the `optionDef` these options are not only used together but they share the same meaning and they result in a similar command line when passed to the tool, e.g. all pre-processor options that are defines. The *name* attribute can be referenced from the `ToolSet` document type. Thus the specific handling of this group of options can be defined there. The *require* attribute can be used to bind the evaluation of this group of options to a condition.

4 The `condValue` element contains the actual value of an option, i.e. the string that finally appears in the tool command line. This can be specified through its various attributes. Again the evaluation of this option can be bound to a logical condition using the *require* attribute.

5 With the *path* and *pathRef* a path can be constructed, e.g. for include directories. The *pathRef* attribute references a symbolic name defined in the `PathSet` document.

6 The *grpref* attribute can be used to reference a `propertyGroup` as defined in the `ConfigSet` document, e.g. to reference all elements of this `propertyGroup` at once with just one line.

7 The *valueRef* attribute references the value of one `property` defined in `ConfigDef` document.

8 The *value* attribute can take up an arbitrary string. In addition to that the following keywords are allowed: `[name]`, `[value]` and `[valname]`. Using these keywords requires the presence of the *valref* attribute which references `property`. If one of these keywords is found it is replaced by the value of a certain attribute from the `ConfigSet` document. `[value]` is replaced by the *value* of the `valueDef` element, `[valname]` is replaced by the *valname* of the `valueDef` element, `[shortname]` is replaced by the *shortName* of the enclosing `propertySet` element, and `[name]` is replaced by the *name* of the enclosing `propertySet` element.

3.5.2. Sample `OptionSet` Document

The example defines two groups of options named `debug_opt` and `autoname_opt`.

One is used to influence the command line of the compiler depending on whether the debug or non-debug version is built, e.g. whether the command line contains pre-processor definitions. The other defines a directory name pattern.

Figure 3.13. Sample `OptionSet` Document

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE optionSet SYSTEM "bsbDocs.dtd">
<optionSet description="Example - Common Options" name="options">
  <options>
    <optionDef description="debug options" name="debug_opt">           (1)
      <condOption name="define">                                       (2)
        <condValue require="DEBUG_MODE==1" value="NEED_HELP"/>
      </condOption>
    </optionDef>
    <optionDef description="auto name options" name="autoname_opt">
      <condOption name="out_dir">                                      (3)
        <condValue value="out"/>
        <condValue valRef="DEBUG_MODE" value="[valname]"/>             (4)
      </condOption>
    </optionDef>
  </options>
</optionSet>
```

1. This defines the option group `debug_opt`. This group is referenced by the `SourceSet` document. It contains one subgroup `define`.
2. The option group `define` contains only one option. This option is a string with the value `NEED_HELP`. Depending on a condition this option is evaluated. The condition references the property `DEBUG_MODE` defined in the `ConfigDef` document.
3. This defines the option (sub)group `out_dir`. This group specifies a construction rule for directory (or file) names. All `condValue` elements will become a part of the constructed name in their order of appearance. This construction rule is referenced by the `PathSet` document.
4. The `condOption` `out_dir` is constructed of two parts: the fixed string “out” and the keyword `[valname]`. This keyword is replaced by the value of the *valname* attribute of the `valueDef` element as defined in the `ConfigSet` document, i.e. if the `ConfigDef` document specifies the value 1 (“debug”) for `property DEBUG_MODE` then `[valname]` is replaced by the string “_db”. The resulting string in that case is “out_db”.

3.5.3. `OptionSet` information in the makefile

The following makefile snippet illustrates how various parts of the previously defined example `OptionSet` document are represented in the makefile. Please note, that this snipped may not be found exacly like this the makefile. The relative order is preserved but some lines may have been stripped out for readability.

Figure 3.14. `OptionSet` information in the makefile

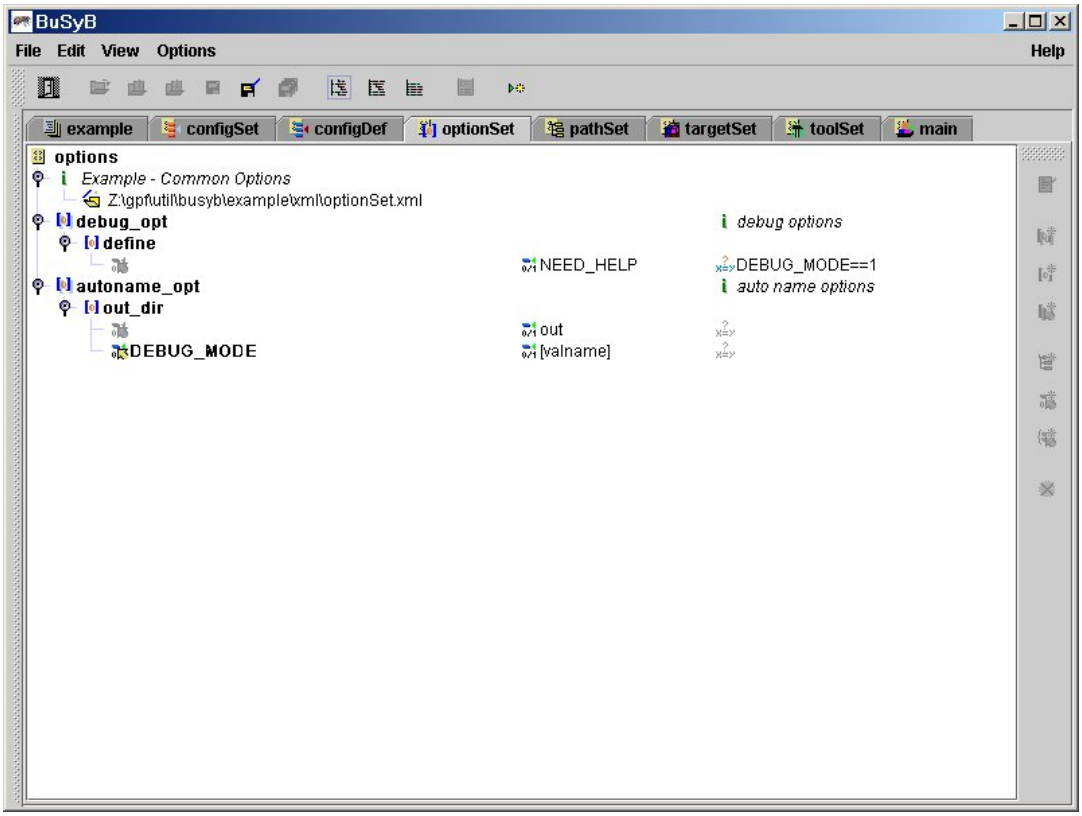
```
out/out_db/lib/main.obj: \                                             (1)
        src/main/main.c
        cl /c /MLd /Z7 /Od /GX /W3 \
        /DNEED_HELP \                                                  (2)
        /Isrc/inc \
         /Foout/out_db/lib/ \
        $<
```

1. The path where the object file will be created contains the pattern `out_db`. This has been automatically created according to the option `out_dir`.
2. The pre-processor definition `NEED_HELP` will be passed to the C compiler according to the option `define`.

3.5.4. Sample `OptionSet` editing with BuSyB

The following figure shows how the previous example is presented by BuSyB. Editing the `OptionSet` is straight forward. When selecting an element new subelements can added via the File menue or via dedicated buttons. Elements can be deleted the same way. The value of attributes can be changed similarly, too.

Figure 3.15. Sample `OptionSet` editing with BuSyB



3.6. Module Properties

The `SourceSet` Document

The `SourceSet` document is designed to contain all those information that are specific to one software module. Usually a number of `SourceSet` documents exist - one for each module - reflecting the software architecture of the system. A `SourceSet` document contains the names of the source files and defines options that are relevant to these files.

All Files in a SourceSet are handled equally

The rational behind grouping source files in a `SourceSet` is that they all belong together from a software architecture point of view. In addition to that it is assumed that their handling during the build process is roughly the same, e.g. the same tools are used, these tools are called with the same options and all files are composed into the same target.

It is possible to bind the presence of a file to a condition, i.e. to include it in or exclude it from the build process depending on a Main Property with the "require" attribute. It is nevertheless not possible to process two file from the same `SourceSet` with different tools. And it is only partially supported to set individual build options for each file.

If all files of a `SourceSet` document share the same extension, e.g. `*.c` or `*.java`, it is possible to leave out that extension. Instead it will become part of a `sources` element of a target definition in the `TargetSet` document.

3.6.1. DTD for `SourceSet` Documents

The `SourceSet` document may contain the module specific `settings` which is a list of `options`. These elements are defined in the `OptionSet` part of the DTD. So for the definition of module specific options, the same mechanism as described for the `OptionSet` can be used.

Figure 3.16. DTD for `SourceSet` Documents

```
<!ELEMENT sourceSet     ( comment*, settings, sourceDirs, sourceFiles )(1)>
<!ATTLIST sourceSet       name        CDATA #REQUIRED
                          description CDATA #REQUIRED>
<!ELEMENT sourceDirs    ( comment*, srcDir, expDir? )>                  (2)
<!ELEMENT sourceFiles   ( comment*, source* )>
<!ELEMENT srcDir        ( comment* )>
<!ATTLIST srcDir          pathRef     CDATA #IMPLIED
                          path        CDATA #IMPLIED>
<!ELEMENT expDir        ( comment* )>
<!ATTLIST expDir          pathRef     CDATA #IMPLIED
                          path        CDATA #IMPLIED>
<!ELEMENT source        ( comment* )>                                   (3)
<!ATTLIST source          require     CDATA #IMPLIED
                          localFlags  CDATA #IMPLIED
                          name        CDATA #REQUIRED>
```

1. The `SourceSet` element contains `sourceFiles`, `sourceDirs` and `settings` elements. It can be attributed with a *name* and a *description*.
2. The `sourceDirs` element defines two directories, the directory (`srcDir`) where the source files, specified later in this document can be found and an export directory (`expDir`). This directory may contain the files representing the "interface" of the modul. The directory names can either be entered directly or reference a path defined in the `PathSet` document. The latter of which is recommended. [5]
3. The `sourceFiles` element contains a list of source files (`source`). The *name* attribute contains the name of the file - possibly without its extension. The *require* attribute can contain a condition that conduct the presence of this file in the build process.

3.6.2. Sample `SourceSet` Document

[5] A mechanism using this information is not yet implemented.

This example defines a `SourceSet`, i.e. a set of files, consisting of only one file (`util.c`). The options for the build of this “module” are partly imported from the `OptionSet` document (*util_global_opt*) and partly defined locally (*util_local_opt*). Two directories important (export and source directory) will be defined via references to the `PathSet` document.

Figure 3.17. Sample `SourceSet` Document

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE sourceSet SYSTEM "../bsbDocs.dtd">
<sourceSet description="Example - Module Properties, module Util"
           name="util">
  <settings>
    <options>
      <optionDef description="Imported global settings (from optionSet)(1)"
                 name="util_global_opt">
        <condOption optRef="debug_opt"/>
      </optionDef>                                                    (2)
      <optionDef description="Local settings" name="util_local_opt">  (3)
        <condOption name="include">
          <condValue pathRef="UTIL_INC"/>
        </condOption>
      </optionDef>
    </options>                                                        (4)
  </settings>
  <sourceDirs>
    <srcDir path="" pathRef="UTIL_SRC"/>
    <expDir path="" pathRef="UTIL_INC"/>
  </sourceDirs>                                                       (5)
  <sourceFiles>
    <source name="util.c"/>
  </sourceFiles>
</sourceSet>
```

1 One option group is defined here. It imports (all) the subgroups (`define`) of the option group `debug_opt` from the `OptionSet` document.

2 Another option group is defined here. Here the subgroup `include` is specified locally, i.e. not as a reference.

3 The path `UTIL_INC` is specified here, which is a reference to a path defined in the `PathSet` document.

4 The two module directories are specified here. A directory (`UTIL_SRC`) where its source files are located and one (`UTIL_SRC`), where the header files can be found. Both directories are defined using a reference from the `PathSet` document.

5 Here the files that are part of set `SourceSet` are specified. The example contains only one source file (`util.c`).

3.6.3. `SourceSet` information in the makefile

The following makefile snippet illustrates how various parts of the previously defined example `SourceSet` document are represented in the makefile. Please note, that this snipped may not be found exacly like this the makefile. The relative order is preserved but some lines may have been stripped out for readability.

Figure 3.18. `SourceSet` information in the makefile

```
out/out_db/lib/util_obj/util.obj: \
```

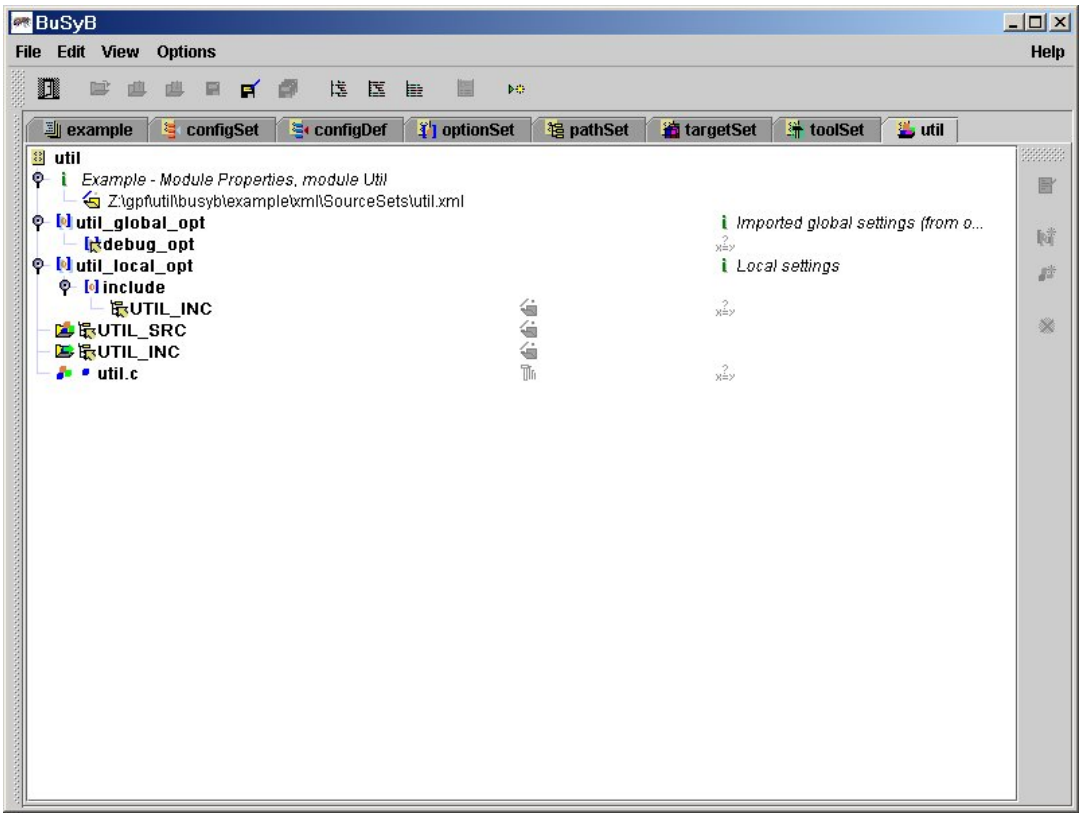
```
        src/util/util.c
        cl /c /MLd /Z7 /Od /GX /W3 \
        /DNEED_HELP \
        /Isrc/inc \                                                    (1)
         /Foout/out_db/lib/util_obj/ \
        $<
```

1 The include directory definition `src/inc` will be passed to the C compiler according to the option `include`.

3.6.4. Sample `SourceSet` editing with BuSyB

The following picture shows how the contents of the previously defined SourceSet document are displayed by BuSyB.

Figure 3.19. Sample `SourceSet` editing with BuSyB



3.7. Tool Properties

The `ToolSet` Document

The `ToolSet` document contains a set of tools, i.e. the software used during the build process to produce the targets. These tools serve different purposes and thus differ heavily in their command line and the supported parameters. Some tools are executables, others are scripts that must be interpreted by an interpreter. Sometime tools must be combined into a tool chain.

3.7.1. DTD for `ToolSet` Documents

Figure 3.20. DTD for `ToolSet` Documents

```
<!ELEMENT toolSet       ( comment*, preambles*, tool+ )>              (1)
<!ATTLIST toolSet         name        CDATA #REQUIRED
                          description CDATA #REQUIRED>
<!ELEMENT preambles     ( comment*, preamble+ )>                      (2)
<!ATTLIST preambles       description CDATA #REQUIRED
                          require     CDATA #IMPLIED>
<!ELEMENT preamble      ( #PCDATA )>
<!ELEMENT tool          ( comment*, command+ )>                       (3)
<!ATTLIST tool            name        CDATA #REQUIRED
                          description CDATA #REQUIRED
                          singleRun   CDATA #IMPLIED>                 (4)
<!ELEMENT command       ( comment*, executable, supports*, args* )>   (5)
<!ATTLIST command         require     CDATA #IMPLIED                  (6)
                          name        CDATA #REQUIRED
                          description CDATA #REQUIRED
                          modifier    CDATA #IMPLIED>                 (7)
<!ELEMENT executable    ( comment*, flag* )>                          (8)
<!ATTLIST executable      pathRef     CDATA #IMPLIED
                          path        CDATA #IMPLIED
                          toolRef     CDATA #IMPLIED>                 (9)
<!ELEMENT supports      ( comment* )>                                 (10)
<!ATTLIST supports        require     CDATA #IMPLIED
                          option      CDATA #REQUIRED                 (11)
                          flag        CDATA #IMPLIED>
<!ELEMENT args          ( comment* )>                                 (12)
<!ATTLIST args            require     CDATA #IMPLIED
                          template    CDATA #REQUIRED>                (13)
<!ELEMENT flag          ( comment* )>                                 (14)
<!ATTLIST flag            require     CDATA #IMPLIED
                          valueRef    CDATA #IMPLIED                  (15)
                          value       CDATA #IMPLIED>
```

1 The `SourceSet` element contains `preambles` and `tool` elements. It can be attributed with a *name* and a *description*.

2 Each `preambles` element contains a list of `preamble` elements. A `preamble` element is a line that goes unchanged into the makefile. These lines will be placed at the beginning of the makefile, i.e. right after the the first (or default) target. A `preamble` can be used to express a Make construct that is not (yet) supported by BuSyB. All `preamble` elements within one `preambles` section will be grouped together in the makefile and headed by a comment containing the *description*. If a `preamble` element contains references to any of the build-in BSB_Variables, this variable will be automatically generated and preset near the top of the generated makefile. The *require* attribute can be used to bind the presence of a `preambles` section to a condition.

3 A `tool` describes a piece of software, which is executed during the build on input files to produce output files. Each `tool` element contains a list of `command`s. Normally this list contains exactly one `command`. Only in case of modeling a tool chain this list will contain more than one `command` element. A tool chain in this sense means a tool that consist of a number of successive calls to (possibly different) tools producing intermediate files which are of not interest and a final result which is of interest. Each `tool` element can be attributed by a *name* and a *description*.

4 The attribute *singleRun* can be used to describe the collecting nature of a `tool`. When the `tool` is applied to a number of files, e.g. to all files of a `SourceSet`, it is via this attribute that can be controlled whether all files are passed to the `tool` in one call or in each file individually in a number of successive calls. To mark a `tool` as “collecting” *singleRun* must be set to *yes* otherwise (the default) the `tool` will be run for each file individually. A good example for a “collecting” `tool` is an archiver

which produces a library from a number of object files with just one call. An example for a "non-collecting" `tool` is a C compiler which produces a number of object files from a number of C files in a series of successive calls.

5 The `command` element contains one `executable` element and lists of `supports` and `args` elements. It describes one command in a tool chain. It can be attributed with a *name* and a *description*.

6 The attribute *require* can be used to bind the presence of this command in the tool chain to a condition.

7 The attribute *modifier* can be used to add a Make modifier to the command in the makefile. The two commonly used Make modifiers are "`@`" and "`-`". The "`@`" tells make to suppress echoing the command. The "`-`" is for ignoring errors.

8 The `executable` element describes a physically (i.e. on the hard disk) present software (executable or script). It contains a possibly empty list of `flag` elements. With the attributes *path* and *pathRef* the physical location of this software can be defined.

9 The attribute *toolRef* can be used to describe that this "`executable`" is not executable but rather a "script". It thus can not be run stand alone but instead needs to be interpreted by some "interpreter", e.g. Perl, Java or Make. In that case the *toolRef* attribute references a `tool` (i.e. "the interpreter") defined elsewhere in the `ToolSet` document.

10 The `supports` element describes a command line parameter that is supported (or provided) by the tool. This element is especially suitable to describe command line arguments that can appear several times on the command line, e.g. -D or -I for a C compiler. The attribute *require* can be used to bind the presence of this element on a condition.

11 The attributes *flag* and *option* control how the `supports` element is represented on the command line. The basic idea is that each appearance of this parameter on the command line consists of two parts. First comes the (possibly empty) *flag* part which is typically a "-" or "/" sign followed by a letter. After that a string (the *option*) follows which is specific to this instance of the parameter on the command line. When looking at `-DDEBUG` the *flag* would be `-D` and the *option* would be `DEBUG`. A more detailed example can be found in the `ToolSet` document.

12 The `args` element describes how the command line of the tool is composed. The presence of certain parts and their placement (beginning, middle, end) can be controlled. The attribute *require* can be used to control the validity of this element by a condition.

13 The attribute *template* defines a template for the command line. The value of this attribute can be a mix of both plain text and some keywords. Keywords are things between brackets, i.e. the text between a "[" and a "]" sign is a keyword. The following keywords exist `[dest]`, `[dest_dir]`, `[dest{.ext}]`, `[source]`, `[source*]`, `[source_dir]`, `[source{.ext}]`, `[localFlags]`, `[cmdFile]`. Each of these keywords will be replaced when the command line is generated into the makefile. `[dest]` will be replaced with the target of the rule, i.e. the file that should be created. `[dest{.ext}]` is handled similarly, however, the target-file extension (i.e. the targets type) will be replaced with the specified string. `[dest_dir]` will be replaced with the directory of the target. `[source]` will be replaced with the source of the rule, i.e. the input file. `[source{.ext}]` is handled similarly, however, the source-file extension (i.e. the source type) will be replaced with the specified string. `[source*]` will be replaced with the list of all input files, i.e. for a collective tool like a linker. `[source_dir]` will be replaced with the directory of the input file. `[localFlags]` will be replaced with the value of the attribute *localFlags* as defined in the `SourceSet` or `TargetSet` documents. The keyword `[cmdFile]` will be replace by the name of temporarily created file. All text or other keywords that can be found right after `[cmdFile` and before the closing "`]`" will be echoed into this temporary file first, e.g. `-@[cmdFile -DDEBUG -c [source]]` would result in a command line that has two parts, the first one echoing the *-DDEBUG -c "the_source_file"* into a file and the second part calling a tool with *-@"the_temp_file"*. This syntax is necessary for tools which are controlled

by command files. Apart from these keywords any text found between brackets is interpreted as the name (*option*) of a former defined supports element. The replacement is different here. For each `condValue` found in any `condOption` with that name, i.e. as defined in either `SourceSet`, `OptionSet` or `TargetSet` documents one pair of the *flag* and the the `condValue` is created. A more detailed example can be found in the `ToolSet` document.

14 The `flag` element describes a command line parameter of the tool. In contrast to the `supports` element this element is for command line parameters that take no additonal arguments, e.g. -c for a C compiler. The attribute *require* can be used to bind the presence of this element to a condition. Though more than one `flag` element is allowed here, only one should be valid at a given time. That means in case of multiple `flag` elements each of them should be restricted by mutually exclusive conditions in the *require* attribute. The value of the `flag` element will at the beginning of the command line, before anything that is specified in the *template* attribute of the `args` element.

15 The attributes *value* and *valueRef* take the actual string of which the command line parameter consists. Whereas *value* takes an arbitrary string, *valueRef* can be used to reference a `condValue` as defined in either `SourceSet`, `OptionSet` or `TargetSet` documents.

3.7.2. Sample `ToolSet` Document

Figure 3.21. Sample `ToolSet` Document

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE toolSet SYSTEM "bsbDocs.dtd">
<toolSet description="Example - Tool Properties" name="tools">
<preambles description="Some rules for directory creation">            (1)
  <preamble>
allOutDirs: $(BSB_OUT_DIRS_ALL)                                       (2)
  </preamble>
  <preamble>
$(BSB_OUT_DIRS_ALL):;mkdir.exe -p $@
  </preamble>
</preambles>
  <tool description="C-Compiler" name="CC">                            (3)
    <command description="MS C-Compiler" name="MS_CC">                 (4)
      <executable path="cl">                                           (5)
        <flag value="/c /ML /O2 /GX /W3"
              require="DEBUG_MODE==0"/>
        <flag value="/c /MLd /Z7 /Od /GX /W3"                          (6)
              require="DEBUG_MODE==1"/>
      </executable>
      <supports flag="/D" option="define"/>                            (7)
      <supports flag="/I" option="include"/>
      <args template="[define] [include] /Fo[dest_dir]/ [source]"/>    (8)
    </command>
  </tool>
  <tool description="Object librarian, archiver" name="AR" singleRun="yes">
    <command description="MS Archiver" name="MS_AR">
      <executable path="link">
        <flag value="/lib" />
      </executable>
      <args template="/OUT:[dest] [source*]"/>
    </command>
  </tool>
  <tool description="Object linker" name="LNK" singleRun="yes">        (9)
    <command description="MS standard linker" name="MS_LNK">
      <executable path="link">
        <flag value="/debug /PDB:NONE /incremental:no /subsystem:console"
              require="DEBUG_MODE==1"/>
        <flag value="/incremental:no /subsystem:console"
              require="DEBUG_MODE==0"/>
      </executable>
```

```
      <args template="/OUT:[dest] [source*]"/>
    </command>
  </tool>
</toolSet>
```

1 A group of preambles consisting of two lines is defined here. A makefile comment (containing the description) and one line for each preamble will be generated into the makefile.

2 This preamble resembles one line of text that is generated into the makefile. It defines a Make rule.

3 One tool (the C compiler `CC`) is defined here. This tool consists of exactly one command.

4 One command (the Microsoft™C compiler `MS_CC`) is defined here.
The name of the compiler executable is `cl`. There is no full path given (with a *pathRef* attribute). So the the call of the compiler will depend the on correctness of the `PATH` environment variable.

6 Two flags are defined here. Since only one of them can be active at given time they are restricted by mutually exclusive conditions on the value of the `DEBUG_MODE` property in the *require* attribute.

7 One supported command line parameter is defined here. This parameter is constructed of a pair of *flag* and *option* attributes. For each `condValue` found in the `condOption` named `define` in `OptionSet` document this pair will be constructed.

8 Here the complete command line for the tool is created as a template. It is constructed of white space (which will appear unchanged in the makefile) and four blocks containing a keyword or to be more precise something in brackets. At first the result of the transformation of the two `supports` elements `[define]` and `[include]` defined above will be placed on the command line. After that comes the string `/Fo` followed by the ouput file directory and the string `/`. Last is the input file. How this template is filled can be seen in the `ToolSet` part of the makefile.

9 Another tool (the linker `LNK`) is defined here. In contrast to the compiler this is a collective tool, i.e. applied on a number of files it will only be called once. Therefore the attribute *singleRun* is set to `yes`.

3.7.3. `ToolSet` information in the makefile

The following makefile snippet illustrates how various parts of the previously defined example `ToolSet` document are represented in the makefile. Please note, that this snipped may not be found exacly like this the makefile. The relative order is preserved but some lines may have been stripped out for readability.

Figure 3.22. `ToolSet` information in the makefile

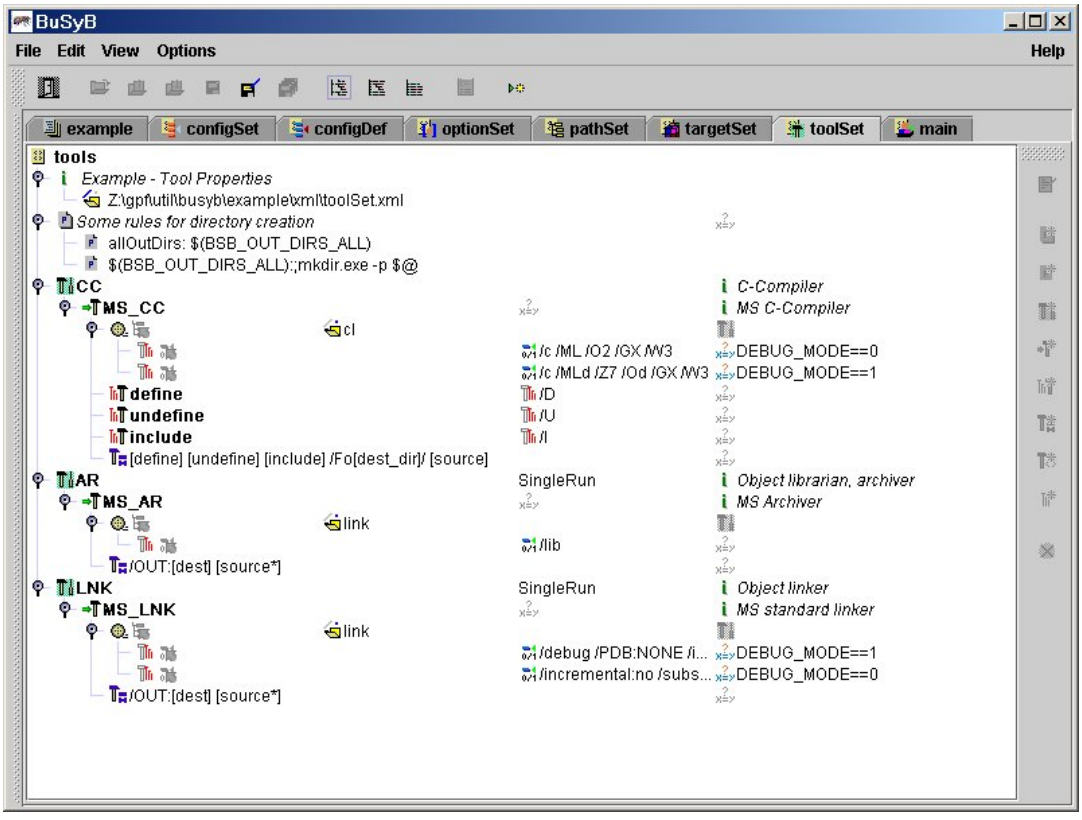
```
out/out_db/lib/util_obj/util.obj: \
        src/util/util.c
        cl /c /MLd /Z7 /Od /GX /W3 \                                    (1)
        /DNEED_HELP \                                                   (2)
        /Isrc/inc \                                                     (3)
         /Foout/out_db/lib/util_obj/ \                                  (4)
        $<                                                              (5)

out/out_db/hello.exe:  \
        out/out_db/lib/main.obj \                                       (6)
        out/out_db/lib/util.lib
        link /debug /PDB:NONE /incremental:no /subsystem:console \
        /OUT:out/out_db/hello.exe \
        $^                                                              (7)
```

1. The tool `cl` is called with the parameters `/c /MLd /Z7 /Od /GX /W3`.
2. The command line starts with `/DNEED_HELP` which is the translation of all *define* options (only one is present coming from the `OptionSet` document).
3. The next block on the command line is `/Isrc/inc`, which is the translation of all *include* options (only one is present coming from the `SourceSet` document).
4. Then the directory of the output file prefixed with `/F` and appended with `/` appears on the command line.
5. The last element on the command line is the input file (`[source]`). It does not appear as plain text but instead the automatic Make variable `$<` is used. This variable will be translated to `src/util/util.c` by Make.
6. The linker is a collective tool. It takes a list of input files (`out/out_db/lib/main.obj` and `out/out_db/lib/util.lib`) and produces one output file with just one call.
7. The last element on the command line is the list of input files (`[source*]`). It does not appear as plain text but instead the automatic Make variable `$^` is used. This variable will be translated to `out/out_db/lib/main.obj` and `out/out_db/lib/util.lib` by Make.

3.7.4. Sample `ToolSet` editing with BuSyB

The following picture shows how the contents of the previously defined `ToolSet` document are displayed by BuSyB.

Figure 3.23. Sample `ToolSet` editing with BuSyB



3.8. Path Properties

The `PathSet` Document

The `PathSet` document type is designed to contain all path information. File paths, input and output directories, tool locations and so on should be specified here. The DTD allows to specify single files as well as directory trees. Each node - file or directory - can be given a symbolic name. These names can then be referenced from almost every other document type.

Paths traversing more than directory level, i.e. containing slashes or backslashes, can either be specified recursively or entered using slashes and/or backslashes. Both characters are understood as a delimiter. They are converted into a single internal representation. The paths in generated makefile will only use slashes.

Both absolute paths as well as relative paths can be entered. Relative paths are always relative to the physical location of the `PathSet` document. Relative paths in the `PathSet` will result in relative paths in the makefile, absolute paths will stay as they are specified. Using relative path information is necessary to cope with clearmake. Part of clearmake's rebuild rules is the comparison of the build command with which an “object” is built. If this command differs - also if it differs “only” in the path of a certain file or tool - the “object” is rebuild. Given this behavior and the fact that it is possible to mount ClearCase views onto different drives it is clear that relative paths are necessary to avoid rebuilds.

Use of relative paths

When creating a makefile into a directory different from the location of the `PathSet` document all relative paths in the makefile will be adjusted. This is done by adding or substracting the “difference” between theese two directories, e.g. when generating the makefile into a subdirectory of the one where the `PathSet` document lies ../ will be added to all paths in the makefile. From that follows that makefiles are not relocatable when using relative path information.

3.8.1. DTD for `PathSet` Documents

Figure 3.24. DTD for `PathSet` Documents

```
<!ELEMENT pathSet       ( comment*, pathTree+ )>                        (1)
<!ATTLIST pathSet         name        CDATA #REQUIRED
                          description CDATA #REQUIRED>
<!ELEMENT pathTree      ( comment*, path+, pathInc*, pathTree* )>       (2)
<!ATTLIST pathTree        name        CDATA #REQUIRED
                          description CDATA #REQUIRED>
<!ELEMENT pathInc       ( comment* )>                                   (3)
<!ATTLIST pathInc         require     CDATA #IMPLIED
                          includePath CDATA #IMPLIED
                          includeFile CDATA #REQUIRED
                          description CDATA #REQUIRED>
<!ELEMENT path          ( comment* )>                                   (4)
<!ATTLIST path            require     CDATA #IMPLIED
                          autoNamed   CDATA #IMPLIED                  (5)
                          value       CDATA #REQUIRED>
```

1 The `PathSet` element contains a list of `pathTree`s. It can be attributed with a *name* and a *description*.

2 A `pathTree` element can recursively contain other `pathTree`s. This is how subdirectories can be specified. The attribute *name* contains the symbolic name by which this node can be referenced.

3 If a valid `pathInc` element is given, the specified file - which must be a valid PathSet document - will be included as a set of pathTrees. The *includeFile* attribute may contain an absolute or relative path. If relativ, it will be evaluated as relative to a specified node in the current `PathSet` document (the *includePath* attribute is present), or to the location of the including `PathSet` document. absolute or relative path. If *includeFile* is absolute, the *includePath* attribute will be ignored. The *description* attribute should contain a note on the included `PathSet`. The *require* attribute controls whether the specified file will be included dependent on the current configuration settings.

4 The `path` element describes a file system node. The *value* attribute is the name of the file or directory. The *require* attribute can contain a logical condition.

5 The *autonamed* attribute can be used to reference a name construction rule defined in the `OptionSet` document.

3.8.2. Sample `PathSet` document

This example defines one root directory. It does so using a relative path. The physical location of this root directory is in the same directory as the `PathSet` document. Its name is `example` and it contains 2 subdirectories - `out` and `src`. These directories again have subdirectories. Each `pathTree` has been given a symbolic name (`EXAMPLE`, `OUT`,`SRC`,...), this allows to reference each node from within other documents.

Figure 3.25. Sample `PathSet` document

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE pathSet SYSTEM "bsbDocs.dtd">
<pathSet description="Example - Path Properties" name="path">
  <pathTree description="complete directory tree" name="EXAMPLE">
    <path value=".."/>
    <pathTree description="output directory tree" name="OUT">
      <path value="out"/>
      <pathTree description="autonamed output directory"
                name="TARGET_OUT">
        <path value="" autoNamed="out_dir"/>                          (1)
        <pathTree description="libraries" name="OUT_LIB">
          <path value="lib"/>
          <pathTree description="object directory for module Util"
                    name="OUT_OBJ_UTIL">
            <path value="util_obj"/>
          </pathTree>
        </pathTree>
      </pathTree>
    </pathTree>
    <pathTree description="source directory tree" name="">            (2)
      <path value="src"/>
      <pathTree description="source dir for module Main" name="MAIN_SRC">
        <path value="main"/>
      </pathTree>
      <pathTree description="source dir for module Util" name="UTIL_SRC">
        <path value="util"/>
      </pathTree>
      <pathTree description="export dir for module Util" name="UTIL_INC">
        <path value="inc"/>
      </pathTree>
    </pathTree>
    <pathTree description="tool directory" name="TOOLS">
      <path value="tools"/>
    </pathTree>
  </pathTree>
</pathSet>
```

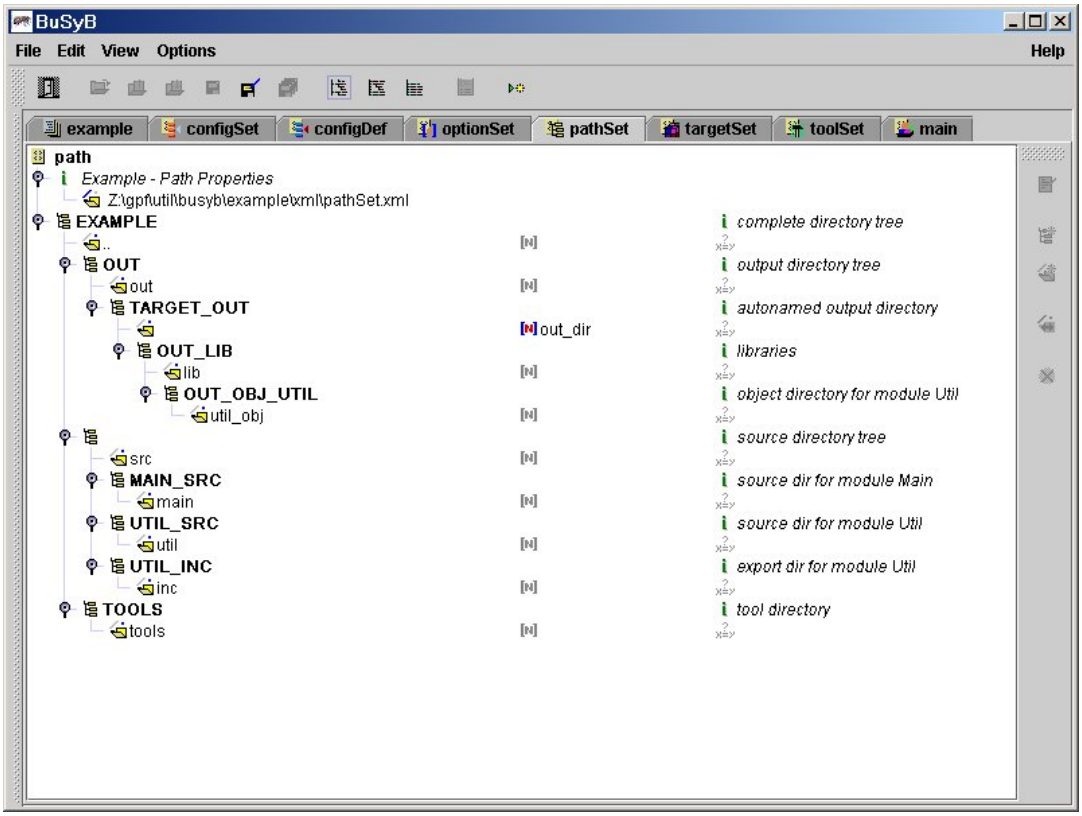
1 Here the name of the directory is not specified directly. Instead a name construction rule is referenced from `OptionSet` document. There the construction of the name depending on (some) Main Properies is defined.

2 Omitting the *name* attribute or entering an empty *name* is allowed. This makes sense for a directory which is never referenced itself but its subdirectories are.

3.8.3. Sample PathSet editing with BuSyB

The following figure shows how the previous example is presented by BuSyB. Editing the `PathSet` is straight forward. When selecting an element new subelements can added via the File menue or via dedicated buttons. Elements can be deleted the same way. The value of attributes can be changed like this, too.

Figure 3.26. Sample PathSet editing with BuSyB



3.9. The BuSyB Command Line

BuSyB can be invoked in two major modes: Edit Mode and Generation Mode. In Edit Mode a GUI Editor allows to edit the XML files. In Generation Mode the contents of the XML files are processed and a makefile is generated. In addition, it is possible to generate Makefiles for selected targets from inside the Target-Editor.

Makefiles generated by BuSyB rely on the existence of some special variables, decribing certain environment properties. These variables may be present in the current execution en-

vironment, or can be defined in the Pragma-Section of the `ToolSet` document. Currently used variables are:

- *BSB_MKDIR* specifies a command to create a directory. It is used, when BuSyB is invoked with the commandline switch -genOutDirs to automatically create needed target directories.
- *BSB_REMOVE* specifies a command to remove a file. It is used, when BuSyB creates the clean-rule commands for each target.
- *BSB_ECHO* specifies a command to dump text into a file. It is used, when BuSyB is directed to generate a command-file by specifying the [cmdFile] keyword in the tools argument-template attribute.

The command line of BuSyB is the following:

`busyb.jar` [-i | -ini *`Inifile`*] [-c | -cfg *`ConfigDefFile`*] [-g | -gen *`target`*] [-out *`Makefile`*] [-l] [-err { a | e | l | x }] [-rpt { 0 | 1 | ... }] [-val] [-var { all | ref }] [-directout] [-softclean] [-settings] [-?]

- `-ini` with this option an Ini-File can be selected. Alternatively option `-i` starts a GUI dialog that allows to browse the file system for an Ini-File. If omitted, BuSyB looks for a file named “bsb.ini” in the current directory.
- `-cfg` with this option a `ConfigDef` document can be selected. Alternatively option `-c` starts a GUI dialog that allows to browse the file system for a `ConfigDef` document. This option overrides the the information found in the Ini-File, where the default `ConfigDef` document is defined. If the given `ConfigDef` has a valid *reference* attribute, the specified file determines the set of documents (`ConfigSet`, `PathSet`, `TargetSet`, etc.) to be used for this specific configuration; in this case, commandline selection of an Ini-File can be omitted.
- `-g` with this option BuSyB is started in Generation Mode. A makefile will be generated for the first target that is found in the `TargetSet` document. With option `-gen` an alternative target can be chosen. If neither of these options is present BuSyB starts in Edit Mode.
- `-out` with this option the name and the location of the makefile that will be generated can be specified.
- `-l` with this option all targets defined in the `TargetSet` document will be listed. A makefile can be generated for each of these targets with option `-gen`.
- `-rpt` with this option the reporting level of the tool can be controlled. Reporting can be disabled (0), enabled (1) for all basic actions and enabled (2) for all actions (verbose).
- `-err` with this option both the error handling and the error reporting level of the tool can be controlled. Verbose error reporting can be turned on for (a)ll errors or only for (e)xternal errors, i.e. things that are out of the responsibility of BuSyB like file system errors. With (l)ong - a full stack trace can be additionally enabled. With early e(x)it - BuSyB aborts on any error otherwise it tries to handle the error.
- `-val` with this option document validation can be turned on, i.e. all XML documents are validated agains the DTD. For this to work correctly the XML documents must

contain “Document Type Declaration”, i.e. reference to the DTD.

- `-var` with this option it can be controlled if BuSyB generates additional variables into the makefile. These variable will contain the content of some BuSyB internal variable. The generation can be turned on for (all) variables or only for those variables which are (ref)erenced in the preamble section of the `ToolSet` document. Currently BuSyB is able to generate the following variables:

  - *BSB_OUT_DIRS_ALL* contains a list of all directories needed by the actual Makefile to generate the target files.
  - *BSB_TARGETS_ALL* contains a list of all target files (including path information) that the current Makefile may generate.
  - *BSB_TG_TYPE_{type}* contains a list of all target files (including path information) of the given {type} that the current Makefile may generate.
  - *BSB_TG_NAME_{name}* contains a list of all files (including path information) that the current Makefile may generate when exetuting the make-rules for the target with the given {name}.
  - *BSB_SOURCES_ALL* contains all files that are target sources.
  - *BSB_SOURCES_ONLY* contains all files that are target sources, but not targets themselves.
  - *BSB_TOOLS_ALL* contains all tools (exes, scripts) for the generated makefile.
  - *BSB_PATHES_ALL* contains name and full path for all PathSet-nodes.
  - *BSB_PROPS_ALL* contains name and value for all ConfigDef-Properties.
  - *BSB_PATH_{name}* contains full path for named PathSet-node.
  - *BSB_PROP_{name}* contains value for named ConfigDef-Property.

- `-directout` turns off in-memory construction of BuSyB output, and lets BuSyB write directly to disc (the old way). This implies the creation of an intermediate file.
- `-softclean` disables the creation of rigid clean rules, so that only the immediate contributers to a target will be removed by the targets clean rule.
- `-settings` displays the state of some internal settings, and the current values of important switches (e.g. defaults for validating, clean rule generation, etc.).
- `-?` (and any invalid option) shows the BuSyB version information and a summary of valid commandline options.

Appendix A. The complete Makefile

This appendix contains the Makefile which has been generated by BuSyB from the XML files presented in the chapter “Using BuSyB”.

Figure A.1. The complete Makefile

```
################################################
##  MAKEFILE FOR           hello
##  DEFINED IN TARGETSET   targets
##                         Z:/gpf/util/busyb/example/xml/targetSet.xml
##  BASED ON CONFIGURATION configuration
##  USING TOOLSET          tools
##  generated              3/23/04 3:15 PM
##  by                     BuSyB Version 1.0.1
##  for                    DTD Version   1.15
################################################


BSB_OUT_DIRS_ALL = out/out_db \
                   out/out_db/lib \
                   out/out_db/lib/util_obj


BuSyB-DefaultTarget: hello


# PREAMBLE: Some rules for directory creation

allOutDirs: $(BSB_OUT_DIRS_ALL)


$(BSB_OUT_DIRS_ALL):;mkdir.exe -p $@




# TargetName=main
#
# TargetType=obj
# TargetDir=out/out_db/lib/
#
# Using Tool: CC
#       mode: One to One
#    command: MS_CC
#   modifier: n/a
#     descr.: MS C-Compiler
#      exec.: cl /c /MLd /Z7 /Od /GX /W3
#
# SOURCES:
# src/main/main.c
#
# RESULTS:
# out/out_db/lib/main.obj
#
# RULES:

out/out_db/lib/main.obj: \
        src/main/main.c
        cl /c /MLd /Z7 /Od /GX /W3 \
        /DNEED_HELP \
        /Isrc/inc \
         /Foout/out_db/lib/ \
        $<


main: \
```

```
        out/out_db/lib/main.obj

clean_main:
        $(BSB_REMOVE) out/out_db/lib/main.obj



# TargetName=util
#
# TargetType=obj
# TargetDir=out/out_db/lib/util_obj/
#
# Using Tool: CC
#       mode: One to One
#    command: MS_CC
#   modifier: n/a
#     descr.: MS C-Compiler
#      exec.: cl /c /MLd /Z7 /Od /GX /W3
#
# SOURCES:
# src/util/util.c
#
# RESULTS:
# out/out_db/lib/util_obj/util.obj
#
# RULES:

out/out_db/lib/util_obj/util.obj: \
        src/util/util.c
        cl /c /MLd /Z7 /Od /GX /W3 \
        /DNEED_HELP \
        /Isrc/inc \
         /Foout/out_db/lib/util_obj/ \
        $<


# TargetName=util
#
# TargetType=lib
# TargetDir=out/out_db/lib/
#
# Using Tool: AR
#       mode: Single Run
#    command: MS_AR
#   modifier: n/a
#     descr.: MS Archiver
#      exec.: link /lib
#
# SOURCES:
# out/out_db/lib/util_obj/util.obj
#
# RESULTS:
# out/out_db/lib/util.lib
#
# RULES:

out/out_db/lib/util.lib:  \
        out/out_db/lib/util_obj/util.obj
        link /lib \
        /OUT:out/out_db/lib/util.lib \
        $^


util: \
        out/out_db/lib/util.lib

clean_util:
        $(BSB_REMOVE) out/out_db/lib/util.lib
        $(BSB_REMOVE) out/out_db/lib/util_obj/util.obj



# TargetName=hello
#
# TargetType=exe
# TargetDir=out/out_db/
#
```

```
# Using Tool: LNK
#       mode: Single Run
#    command: MS_LNK
#   modifier: n/a
#     descr.: MS standard linker
#      exec.: link /debug /PDB:NONE /incremental:no /subsystem:console
#
# SOURCES:
# out/out_db/lib/main.obj
# out/out_db/lib/util.lib
#
# RESULTS:
# out/out_db/hello.exe
#
# RULES:

out/out_db/hello.exe:  \
        out/out_db/lib/main.obj \
        out/out_db/lib/util.lib
        link /debug /PDB:NONE /incremental:no /subsystem:console \
        /OUT:out/out_db/hello.exe \
        $^


hello: \
        out/out_db/hello.exe

clean_hello:
        $(BSB_REMOVE) out/out_db/hello.exe



clean:  \
        clean_main \
        clean_util \
        clean_hello


## END OF GENERATED MAKEFILE
#################################################
```