Of course, replace version-name by the release version name. You must repeat this operation for all svn module that need to be tagged.

Publishing

Publishing means copying files in the web directory. The sync-to-www.sh moves the documentations in the doc/ subdirectory:

```
/doc
|-- examples
|-- html
|-- pdf
```

Compiler & engines releases must be copied by hand.

10.3 Things to keep in mind

Unless you know exactly what you are doing (and why!), you should:

• **never** commit `u.v.b.userinterface.cli/src/main/java/u/v/b/userinterface/Version.java`: it is modified when building a distribution but these modifications should never make their way into the code repository.

• **never** commit any `.project`, `.classpath` or any other eclipse dot-files. Having local modifications is also often a sign of wrong configuration (but not always). Be careful to never commit these as this will break other developer setup.

• **never** commit `Documents/sphinx-doc/source/conf.py` if you've only changed the version/release information.
10.2 Publishing a distribution

Publishing a distribution must always include the following steps:

1. **compute a new version name**. You must never re-use a previous version name. Never! If you have published a distribution with a huge bug inside, it’s already too late, you need a new version. Commonly, a version has a major part and a minor or revision part. The major corresponds to a new release, and the minor can be changed when you want to distribute a new revision for the same release (e.g. with bug fixes).
2. **build the compiler** using the previously computed version name
3. **build the engines** using the previously computed version name
4. **build the document**. The documentation should always match the distributed software. Never provide outdated documentation. If you can’t update the documentation (it’s a shame), outdated part must be explicitly marked.
5. **tag** all the different parts of the software, script, documents in the SCM (e.g. subversion).
6. **publish**!

The scripts presented in previous paragraphs can help for steps 1, 2, 3. Steps 4, 5 and 6 must be done carefully by hand.

10.2.1 Manual steps

Building the documentation

Building the documentation is simple. 2 documentations can be built and published: APIs & user/dev documentations. As of this writing, only the user/dev documentation is published though, as APIs still need some work (they are based on javadoc & doxygen).

The user/dev documentation are using Sphinx.

First, you need to configure the documentation build to include the correct version number and release name. Edit the file `source/conf.py` to include the matching version/name:

```python
version = '2012.04'
release = '2012.04 (RC3)'
```

Building is as easy as running:

```bash
$ make html latexpdf
```

The targets are self-described and produce static HTML pages and a PDF. The script `sync-to-www.sh` provides an easy way to build and publish the user/dev documentations. The script also creates a tree-hugger-friendly PDF with 2 pages per side and publishes the example files.

Tagging

You must tag all parts (compiler, engines, documentation, distribution scripts), even the ones that have not moved since the previous release, use the `tag` command:

```bash
$ svn tag version-name
```
CHAPTER
ONE
INTRODUCTION
This document starts by introducing the main concepts of the BIP2 language: types, semantics and of course its syntax (see The BIP2 Language). Then, it presents tools used to compile and execute BIP2 programs. The compiler and the engine: their installations and basic usage. As the main use cases involve the generation of C++ code, a dedicated part explains more deeply how to use the C++ code generator of BIP2 (see More about C++ code generator). A step-by-step tutorial shows how to use the main features of the BIP2 language (see Tutorial). Finally, the full language syntax is included as a reference (see BIP 2 Grammar).

1.1 Conventions used in this documentation

1.1.1 Shell commands

Shell commands are preceded by '$':
$ cd /etc/
When a command needs to be executed from within a given directory, this directory is mentioned before the $:
/home/bla/ $ mkdir toto
If a command line is too long, the line is cut by escaping the line ending character:
$ ./bla --this --is="a very long" --command --line --that --is --cut=twice

118 Chapter 10. Developer reference for building and packaging

10 Single archive distribution using single-archive-dist.sh

The -s can be used to skip the compiler compilation, which is responsible for most of the build time. The remaining steps are still executed.
The last parameter, -p, is used to set the cmake build profile used when building the engines. Default is Release, meaning that the code may be optimized and debugging symbols stripped. If you need to build a debugging release, use the Debug profile. Please note that this profile is only used for building the engines, it has nothing to do with the build of the code that may be generated by the bip compiler later.

10.1 The single archive distribution process is as follows:
A new build directory is created for the distribution. Inside this directory, the engines are compiled.

10.1.1 Building the compiler

The bin directory is created for the compiler. Inside, the following rules are created:

• Makefile: to build the compiler
• CMakeLists.txt: to build the compiler

After the compiler has been successfully built, the wrap script continues by copying the resulting artefacts (jar files) and the frontend script that will be used by the users (ie. bipc.sh).

At this point, the bip compiler is ready. The wrap script now compiles the different engines found in the Engines/ directory. The result of the engine compilation is directly a self-contained archive.

10.1.2 What wrap.sh does

After everything has been executed, you can find the distribution in the build/ directory:

• bipc-2012.04.110853-DEV.FILES
• bipc_2012.04.110853-DEV.tar.gz
• BIP-optimized-engine-2012.04.110853-DEV_Linux-i686.tar.gz
• BIP-reference-engine-2012.04.110853-DEV_Linux-i686.tar.gz

The .FILES file should be kept as it contains all the filenames included in the compiler distribution archive. It is useful, as the compiler contains several external dependencies whose versions are not encoded in the distribution version.

10.1.3 Single archive distribution using single-archive-dist.sh

For even easier distribution, the single-archive-dist.sh script can be used. It produces a single archive with the compiler and the engines inside. Installation is only a matter of extracting and running a script that sets up the environment correctly. It relies on wrap.sh and simply rearrange the products of the latter. The script accepts only -r and -v command line parameters, which are exactly the same as for wrap.sh.

The result of running this script is a single archive called bip-full_<ARCH>.tar.gz. It contains the compiler and the engines. It also has a setup.sh script that can be used to setup the environment correctly. By default, the script configures the environment for using the reference-engine, but you can give it any engine (provided it is shipped in the archive) name:
$ . ./setup.sh optimized-engine
Environment configured for engine: optimized-engine
$ . ./setup.sh
Environment configured for engine: reference-engine

118 Chapter 10. Developer reference for building and packaging
10.1 Building a distribution

Building a distribution consists of the building of both the compiler & the different engines. The distribution must meet the following goals:

- contains only the code we want to distribute (i.e. no source code)
- requires as less installation steps as possible for the user
- ability to keep track (version) of distributed files

The distribution scripts involve several steps:

- compute a distribution version number and inject it in all following steps
- compile the compiler: it uses ant/ivy
- compile the different engines (eg. reference engine, optimized engine)
- provide install/setup script
- package compiler & engines in archives to be distributed

All these steps are scheduled from the script called wrap.sh located, as everything related to the distribution, in the distribution directory.

Important: Please be aware that all steps needed in the release process are not automated in scripts. You still need to do manual steps (see below) in order to build a complete release. Failing to do so will lead to incoherence and most probably headaches for solving problems.

10.1.1 Invoking wrap.sh script

The wrap script accepts several command line parameters:

- `-r` When building a revision to release :)  
- `-v` Give the version name instead of it being generated  
- `-p` Build profile for engine (default: Release)  
- `-s` Skip directly to engine building, no compiler  
- `-h` This help

If the `-r` flag is used, then the version used throughout the build uses the pattern YYYY.MM. It has the nice property of being easy to compute, to understand and is always increasing.

The `-v` allows the specification of the full version string. It has priority over `-r`.

If none of `-v` and `-r` are used, then the pattern used is YYYY.MM.HHmmSS-DEV.
2.1 Introduction

Figure 2.1: The three-layered BIP2 representation.

BIP2 (Behavior, Interaction, Priority version 2) is a component-based language for modeling and programming complex systems. In BIP2, a system is represented by:

- **Behavior** specified by a set of components
- A set of **interactions** which defines the possible synchronizations and communications between the components; they are structured in connectors corresponding to a subset of interactions (see Connectors).
- A set of **priorities** used for resolving conflicts between interactions or for defining interaction schedule policies (see Priorities).

With behavior, interactions and priorities we can build hierarchies of complex components called compound or compounds for short. A compound component is composed of a set of components, connectors and priorities (see Compounds).

**Atomic components**, or atoms for short, are the simplest component type (i.e., non-hierarchical) whose behavior is expressed by automata or Petri nets (see Atoms).

In the following, we use the term **component** to refer to either an atomic or a compound component. The ports and variables accessible to other components and connectors define the component interface. Ports are used for component communication in a synchronized manner. Variables store information accessible to priority and transition guard expressions to resolve conflicts and non-determinism.

The BIP2 compiler processes an input file that contains a **package** declaration. In the processed file, a compound component, called **model**, describes the system we want to simulate, analyze, verify or just execute.
2.2 Quick overview of the language

2.2.1 Preliminary notations

In the following sections we describe the main features of the BIP2 language. The language syntax is expressed by a set of derivation rules that observe the following conventions:

- a rule begins with a name followed by the symbol ':=' and one or more terminal and non-terminal rules, e.g.: `non_term := 'term' sub_non_term`
- terminal elements are enclosed in '', e.g. `'terminal'`

Identifiers are used in many contexts to denote package names (package_name), variables (variable_name) etc. In reality, those constructs are expressed by one rule in the grammar, but for readability we refer to them with a descriptive rule synonym. You can find the full grammar in BIP 2 Grammar.

Examples of rules

```
sample_rule := 'some text' another_rule 'some ending text'
another_rule := 'foo bar terminal'
```

2.2.2 Annotations

Annotations offer a mechanism for defining information that are used by tools other than the compiler. The compiler examines the syntax of annotation directives but their content is ignored. BIP2 statements that accept annotations are noted by the following notation:

- accepts annotations

The syntax for the annotations is given below.

**Syntax**

```
annotation := '@' annotation_name [('@' annotation_parameter (',' annotation_parameter)+ ')']
```

**Example**

```
@cpp(foo=bar, obj="foo.o,bar.o")
atom type MyAtom(int x)
...
end
```
2.2.3 Packages
A package is a unit of compilation contained in a single file. It may include other packages with the use directive. In BIP2, a package may contain:

- constant data (see Variables and data types)
- external data types (see Variables and data types)
- external functions (see Variables and data types)
- external operators (see Actions)
- port types (see Port types)
- atom types (see Atoms)
- connector types (see Connectors)
- compound types (see Compounds)

Constants are referenced in type definitions or in the initialization of other constant data. Constant data are visible only within the package that defines them.

Important:
BIP2 permits the declaration of type names used for simple type checking but doesn't support type definitions (classes, structures, etc.). It's the responsibility of the back-ends to really interpret the types (for example, the C++ back-end will map these types to C++ types directly).

Important:
To refer to types declared in other packages, prefix the type name with the name of the package where it is declared (e.g. some.pack.name.SomeAtomType)

Syntax
• accepts annotations
package_definition :=
'package' package_name
('use' package_name)*
data_type*
(extern_function | extern_operator)*
bip_type+
'end'
data_type :=
'extern data type' type_name
['refine' type_name (',' type_name)*]
['as' '"' backend_name '"']
extern_function :=
'extern function' ['type_name'] function_name '(' ['type_name (',' type_name)*'] ')' 
extern_operator :=
'extern oprator' ['type_name'] operator '(' ['type_name (',' type_name)*'] ')' 

2.2. Quick overview of the language
Example

```java
package SomePackage
const data int my_const_int = 42
extern data type my_list
extern function int min(int, int)
extern function print(string)
extern function int get(int i, my_list)
port type Port_t()
port type Port_t2(int i, my_list l)
end
```

2.2.4 Variables and data types

In BIP2, variables are used to store data values. Their declaration consists of a (data) type and a name. For example:

```java
data int x
declares a variable named x of type int. The keyword data is omitted in the declaration of parameters of BIP2 types (i.e. port types, atom types, connector types, and compound types). Constant variables can also be declared in packages using the keyword const data and the initialization operator =. For example:
const data float Pi = 3.1415926
```

at the beginning of a package declares a constant named `Pi` of type `float` with value `3.1415926`.

**Important:** The constant variables of packages are the only ones that can (and must) be initialized when declared. Other types of variables should be initialized after their declaration.

Types of variables are either native or external. Native types are known to the BIP2 compiler and are part of the language. Currently, the supported native types are:

- **bool** for boolean values `false` and `true`
- **int** for integers (e.g. `-100, 0, 32`)
- **float** for floating-point numbers (e.g. `2.7182818`) (for sequences of characters (e.g. `"My name is BIP2\n"`).
- **string** for sequences of characters (e.g. `"My name is BIP2\n"`).

Notice that the type `int` is considered by the compiler as a sub-type of `float` regarding compatibility of types, which means each time the type `float` is accepted, the type `int` is also accepted.

**Important:** The exact encoding (number of bits, range) of the native data types is not specified by the semantics of BIP2. Currently, the specialization is done in the back-ends. Typically, native data types are mapped to the usual types of the target language, e.g. when using the C++ back-end the native types of `bool`, `int`, `float`, and `string` are mapped respectively to the C++ types `bool`, `int`, `double`, and `std::string`.

Notice that constant variables of packages, as well as parameters of components, can be only of a native type.

Besides the predefined native types, additional types can be declared with the keyword `extern`. These types are supposed to be externally defined and present when compiling the generated code. For instance, when using the C++ backend all the external types should be defined in additional C++ files included in the compilation process of the generated code. An example of declaration of an external type named `IntList` can be found below.

8 Chapter 2. The BIP2 Language

9.7.4 Misc

The C++ back-end includes 2 utility templates:

- `traceBip.mtl`: contains the needed queries/templates for injecting in the C++ code back-links to the BIP code. Some templates can be used to drive the GNU Debugger (aka. `gdb`) so that it displays the BIP source code instead of the generated C++ code. This features has been prototyped only and has been put on hold in favor of other developments.
- `gcc.mtl`: used to store everything specific to the GNU Compiler Collection (aka. `gcc`). It is currently nearly empty as it only includes a query for asking the compiler not to raise a warning when a specific variable declaration is never used.

9.8 Tutorial

9.8.1 Debugging a Filter or a Backend

The way the compiler is built and configured by default in eclipse won’t let you use any of your filters or backend. The compiler will load dynamically the classes for your filters/backends provided they are in the java classpath.

9.8.2 Adding a new constraint

You must always follow all the following steps. Do not leave some steps as todo tasks, you will always forget to do them, leading to future bugs, longer misunderstanding, etc.

- add constraint in the meta-model. Choose a name as discriminant as possible. You should include everything possible in the name as the constraint name will also be used in error handling. Better use `ConnectorParameterHasBadType` instead of `BAD_TYPE`.
- commit the generated code corresponding to the new constraint. In the comment, add explicitly that it is only a generated code.
- generate the code. This will create an empty method with a query for asking the compiler not to raise a warning when a specific variable declaration is never used.
- traceBip.mtl: contains the needed queries/templates for injecting in the C++ code back-links to the BIP code. Some templates can be used to drive the GNU Debugger (aka. `gdb`) so that it displays the BIP source code instead of the generated C++ code. This features has been prototyped only and has been put on hold in favor of other developments.
- gcc.mtl: used to store everything specific to the GNU Compiler Collection (aka. `gcc`). It is currently nearly empty as it only includes a query for asking the compiler not to raise a warning when a specific variable declaration is never used.

9.8.3 Adding a new filter or backend

If you want to add a new filter or backend, you should follow all the following steps. Do not leave some steps as todo tasks, you will always forget to do them, leading to future bugs, longer misunderstanding, etc.

- add constraint in the meta-model. Choose a name as discriminant as possible. You should include everything possible in the name as the constraint name will also be used in error handling. Better use `ConnectorParameterHasBadType` instead of `BAD_TYPE`.
- commit the generated code corresponding to the new constraint. In the comment, add explicitly that it is only a generated code.
- generate the code. This will create an empty method with a query for asking the compiler not to raise a warning when a specific variable declaration is never used.
- traceBip.mtl: contains the needed queries/templates for injecting in the C++ code back-links to the BIP code. Some templates can be used to drive the GNU Debugger (aka. `gdb`) so that it displays the BIP source code instead of the generated C++ code. This features has been prototyped only and has been put on hold in favor of other developments.
- gcc.mtl: used to store everything specific to the GNU Compiler Collection (aka. `gcc`). It is currently nearly empty as it only includes a query for asking the compiler not to raise a warning when a specific variable declaration is never used.

9.8.4 Debugging a Filter or a Backend

The way the compiler is built and configured by default in eclipse won’t let you use any of your filters or backend. The compiler will load dynamically the classes for your filters/backends provided they are in the java classpath.

9.8.5 Adding a new constraint

You must always follow all the following steps. Do not leave some steps as todo tasks, you will always forget to do them, leading to future bugs, longer misunderstanding, etc.

- add constraint in the meta-model. Choose a name as discriminant as possible. You should include everything possible in the name as the constraint name will also be used in error handling. Better use `ConnectorParameterHasBadType` instead of `BAD_TYPE`.
- commit the generated code corresponding to the new constraint. In the comment, add explicitly that it is only a generated code.
- generate the code. This will create an empty method with a query for asking the compiler not to raise a warning when a specific variable declaration is never used.
- traceBip.mtl: contains the needed queries/templates for injecting in the C++ code back-links to the BIP code. Some templates can be used to drive the GNU Debugger (aka. `gdb`) so that it displays the BIP source code instead of the generated C++ code. This features has been prototyped only and has been put on hold in favor of other developments.
- gcc.mtl: used to store everything specific to the GNU Compiler Collection (aka. `gcc`). It is currently nearly empty as it only includes a query for asking the compiler not to raise a warning when a specific variable declaration is never used.

9.8.6 Adding a new filter or backend

If you want to add a new filter or backend, you should follow all the following steps. Do not leave some steps as todo tasks, you will always forget to do them, leading to future bugs, longer misunderstanding, etc.

- add constraint in the meta-model. Choose a name as discriminant as possible. You should include everything possible in the name as the constraint name will also be used in error handling. Better use `ConnectorParameterHasBadType` instead of `BAD_TYPE`.
- commit the generated code corresponding to the new constraint. In the comment, add explicitly that it is only a generated code.
- generate the code. This will create an empty method with a query for asking the compiler not to raise a warning when a specific variable declaration is never used.
- traceBip.mtl: contains the needed queries/templates for injecting in the C++ code back-links to the BIP code. Some templates can be used to drive the GNU Debugger (aka. `gdb`) so that it displays the BIP source code instead of the generated C++ code. This features has been prototyped only and has been put on hold in favor of other developments.
- gcc.mtl: used to store everything specific to the GNU Compiler Collection (aka. `gcc`). It is currently nearly empty as it only includes a query for asking the compiler not to raise a warning when a specific variable declaration is never used.
In addition, we also allow the comparison of boolean expressions, which are separated by 'true' operators, boolean operators, and function calls (with returned values). As usual, parenthesis is used to group expressions and enforce a specific evaluation order.

Multiple statements in an action are enclosed in brackets. Computations and data transformations in actions are expressed by C-like syntax statements and expressions. States and data values can be accessed in local and external context using identifiers which are meaningful in C++.

Actions define computations and data transformations. In the context of a component, actions are declared as methods of class components. Methods have the same function name even if they have the same number of arguments and/or different return types. This may be used to increase modularity of the code. For example, in a component, an action 'compute' may be declared with different signatures depending on the context:

- `compute(const int &a)`: Computes the sum of an integer argument.
- `compute(const std::vector<int> &vec)`: Computes the sum of a vector of integers.

There are no specific restrictions in the declaration of prototypes concerning overloading: different prototypes may be defined for the same function names. However, overloading is discouraged to avoid confusion and improve code readability.

Important: The limitation only exists for simplification purposes. It is not supported and should not be used.

Actions can be defined in two ways:

- **Component Actions**: Actions are defined as methods of class components. They are declared using the `@action` stereotype.
- **Component Methods**: Actions can also be defined as methods of class components. They are declared using the `@method` stereotype.

### 9.7.3 CMake

CMake is a widely used build system that allows you to create a system capable of handling project configurations. It is a script which specifies the dependencies, tools, and scripts necessary to build a component. CMake scripts are typically written in a language similar to Unix Make.

#### 9.7.3.1 CMakeLists.txt

The `CMakeLists.txt` file is the primary source of configuration for a CMake project. It is typically located in the root directory of the project and is used to specify the project name, version, and dependencies. The following is an example of a `CMakeLists.txt` file:

```cmake
# Project Configuration
cmake_minimum_required(VERSION 3.10)
project(BIP2)

# Project Targets
add_library(BIP2 INTERFACE)
set(BIP2_INCLUDE_DIR ${CMAKE_CURRENT_BINARY_DIR}/include)
set(BIP2_LIBRARY_DIR ${CMAKE_CURRENT_BINARY_DIR}/library)

# Library Dependencies
find_package(BIP2 REQUIRED)
find_package(Boost REQUIRED)
find_package(OpenMP REQUIRED)

# Library Configuration
add_executable(BIP2 ${CMAKE_CURRENT_SOURCE_DIR}/src/bip2.cpp)
target_link_libraries(BIP2 Bip2 ${Boost_LIBRARIES} OpenMP)

# Build Configuration
install(TARGETS BIP2 RUNTIME DESTINATION bin)
install(TARGETS BIP2 ARCHIVE DESTINATION lib)
installFILES(
    ${CMAKE_CURRENT_SOURCE_DIR}/include/ BIP2_INCLUDE_DIR
    ${CMAKE_CURRENT_SOURCE_DIR}/library/ BIP2_LIBRARY_DIR
)

# Not for deployment
install(TARGETS BIP2 ARCHIVE DESTINATION lib)
install(TARGETS BIP2 ARCHIVE DESTINATION bin)

# Not for deployment
install(TARGETS BIP2 ARCHIVE DESTINATION bin)
install(TARGETS BIP2 ARCHIVE DESTINATION lib)
```

This configuration file sets up the project, defines targets for the component, and specifies dependencies on other libraries. It also includes targets for building and deploying the component.

### 9.7.4 Source Code Generation

BIP2 includes a number of source code generation tools that can be used to automatically generate code to support common tasks. These tools can be used to create C++ code for deployment and other tasks.

- **generateDeploy.mtl**: This template generates the main deployment code that deploys the component. The generated code includes a function that is the entry point for the component. Example:

```cpp
Component* deploy(int argc, char **argv);
```

- **generateType.mtl**: This template generates a file that contains the type definitions for the component. Example:

```cpp
extern data type IntList refine List
```

### 9.7.5 Precompiled BIP Package

This feature will be handled in later version.

---

9.8 Quick overview of the language

Quick overview of the language.

---

9.9 Component instances

Component instances are used to create a component instance of a component. They are declared using the `ComponentInstance` stereotype.

```cpp
ComponentInstance Bip2; // declares an instance of Bip2
```

---

9.10 Parameters

Parameters are used to pass values to a component. They can be declared as parameters of the component instance or as parameters of actions.

```cpp
int argc;
char **argv;
```

---

9.11 Actions

Actions are used to define computations and data transformations. They are declared using the `@action` stereotype.

```cpp
@action void compute(const int &a) {
    int result = a + 1;
}
```

• < : less than
• > : greater than
• <= : less or equal than
• >= : greater or equal than

Arithmetic operators provided below can only be applied to numbers, i.e. int and float data types. They return a value of type int if all the arguments are of type int. They return a value of type float otherwise.

• / : division
• % : modulo
• + : addition or positive sign
• - : subtraction or negative sign
• * : multiplication

Logical boolean operators apply to boolean values only (of type bool), and return boolean values:

• && : logical and
• || : logical or
• ! : logical not

Boolean bitwise operators apply to int only, and return int:

• & : bitwise and
• | : bitwise or
• ^ : bitwise exclusive or
• ~ : bitwise not
• : : logical not

The assignment operator may assign a value to a variable provided that the type of this value is compatible with the type of the variable, that is, if it is of the same type or if it is of a sub-type. Notice that in contrast to previous operators, by default the assignment operator applies also to external types.

= : assignment

Important: The exact behavior data types and corresponding operations is not specified by the semantics (e.g. min/max ranges of integer and floating point types, behavior of overflows, etc.). Currently, the specialization is done in the back-ends (usually by mapping directly BIP2 types and operations to usual types and operations of the target language).

In addition to the predefined operators, external functions can be call provided their prototype is declared, as explained in Variables and data types. We say that a function call matches a prototype if it has the same function name and the same number of arguments, and if its arguments are compatible with the ones of the prototype. We say that a prototype is strictly more precise than another prototype if it has compatible arguments with at least one being a strict sub-type. For example in the following the first prototype is strictly more precise than the third prototype, whereas it is not comparable with the second prototype:

extern function float min(float, int)
extern function float min(int, float)
extern function int min(int, int)

A function call will not compile if one of the following assertions apply:

• it does not match any declared external function prototype ("no-match prototype" error)

9.6.4 u.v.b.backend.cpp

This back-end is the most complex (and used) available in the compiler. It uses both the type model and the instance model to generate a set of C++ source file along with the cmake scripts used to build everything.

The type model is used to generate C++ classes. All these classes inherit from classes in the BIP engine interface.

The instance model is used to create the needed statements and variable creation for the deployment of the system.

Entry points for this back-ends are:

• the GeneratePackage class that is the interface between the java code and the acceleo engine that applies the templates for the generation of classes corresponding to BIP types. From the outside (java world), it is only possible to generate something from a package (i.e. it is not possible to generate simply the C++ code corresponding to an atom type).
• the GenerateDeploy class is the interface between java code and the acceleo engine for the creation of the deploy code.
• the Cmake class is used to generate all the necessary files for cmake to build all the generated code.

More details are given in the separate C++ back-end.

9.6.5 u.v.b.backend.example

This back-end is empty and its only use is to be a starting point for creating new back-end.

9.6.6 u.v.b.backend.tests

All JUnit tests are stored in this module. As for the front-end tests:

• test classes must be named SomethingTests with Something being a descriptive name that does not start with Abstract
• tests resources must be placed in sub-directories of src/tests/resources/. The current convention is to store C++ backend related resources in a cpp/ subdirectory.

9.6.7 acceleo.standalone.compiler

A back-end is a black box that is used for generating something from a BIP-EMF model. Typical lifecycle of a back-end:

• configuration (eg. output directory, optimization level, ...)
• if the back-end is able to generate something from a type model, then it is called with the type model at the end of the compilation process
• if the back-end is able to generate something from an instance model and an instance model has been build during compilation, the it is called with the instance model.

9.7 C++ back-end

The back-end must be fed with both the type model and the instance model. The type model is used to create C++ classes and the instance model to create a deployment script (i.e. creates instances of previously created classes, in a correct order).
In the context of an action, a multiple expressions need to be evaluated in order to return a value.

Example

```plaintext
extern operator IntList ||(IntList, IntList) // allowed but not recommended!
extern operator int ==(IntList, IntList) // allowed but not recommended!
```

Notice that declarations of external comparison operators (==, !=, <, <=, >, >=, <>) are forced to return boolean.

This back-end produces BIP code. It is very simple, as templates are used to translate the textual BIP to the textual BIP.

```
extern operator Complex *(float, Complex) // OK
extern operator Complex *(Complex, Complex) // OK
extern operator *(Complex, Complex) // not valid: missing return type - ERROR!
extern operator Complex *(Complex) // not valid: missing argument - ERROR!
```

The declaration of external comparison operators (==, !=, <, <=, >, >=, <>) are restricted to the following:

```
9.6.3 u.v.b.backend.bip
```

The back-end produces Aseba code. It is highly experimental and does not cover all the BIP language.

```
9.6.2 u.v.b.backend.aseba
```

These declarations should always include a return type, and are limited to the number of arguments a given operator has in the language for native types. For example, in the following code the first two declarations are not permitted and will lead to a compilation error such as:

```
[SEVERE] In /path/to/file/my_bip_file.bip:

38: x = min(0, 0);
39: x = min(0, 0);
40: x = min(0, 0);
41: x = min(0, 0);
```

```
// Ambiguous function call 'min' with parameter(s) of type(s) 'int, int': cannot decide return value
```

```
• 1 for the annotations
• 1 for the port declarations
• 1 for the package
• 4 BIP types (ie. int, float, string, Complex)
```

```
9.6 Back-end
```

```
9.6.1 u.v.b.backend
```

The back-end is used to generate Aseba code. It is highly experimental and does not cover all the BIP language.

```
9.6.4 u.v.b.backend.pipes
```

The back-end is used to generate Aseba code. It is highly experimental and does not cover all the BIP language.

```
9.6.5 u.v.b.backend.aseba
```

The back-end is used to generate Aseba code. It is highly experimental and does not cover all the BIP language.

```
9.6.6 u.v.b.backend
```

The back-end is used to generate Aseba code. It is highly experimental and does not cover all the BIP language.

```
9.6.7 u.v.b.backend
```

The back-end is used to generate Aseba code. It is highly experimental and does not cover all the BIP language.

```
9.6.8 u.v.b.backend
```

The back-end is used to generate Aseba code. It is highly experimental and does not cover all the BIP language.
in Atoms, the data can be directly referenced by its declaration name whereas a connector action referencing a data within a port must use a dotted notation (e.g. `port_name.data_name`).

There is currently only one control flow operation: `if-then-else` with the following syntax:

```java
if { boolean_condition } then
  statement1;
else
  statement2;
fi;
```

The `else` part is optional and may be omitted. The expression `boolean_condition` must evaluate to a boolean value.

### 2.2.6 Port types

Ports are used to synchronize component and convey information in a synchronized manner between the components of a model. The transferred information is accessible via the variables associated with the port. Port types are declared with the `port_type` keyword followed by the port type name and a possibly empty list of accessible variables. The following example declares a port with type `port_t` which can access integer values from the `x` variable:

```java
port type port_t(int x)
```

**Syntax**

- accepts annotations

```java
port_type_definition ::= 'port type' (package_name '.'?)? port_type_name
  '(' data_param_declaration (',' data_param_declaration)* ')'?
```

### 2.2.7 Atoms

Atoms are the simplest components with a behavior described by an automaton or a Petri net extended with data. An `atom` type is declared with the `atom` type directive which contains:

- a possibly empty list of variables for storing data. Data declarations may be exported to become accessible to priorities.
- an optional list of port declarations that may reference variables. Exported ports are accessible to connectors.
- an automaton or a Petri net that defines the behavior of the atom. The behavior is described by a set of transitions that change the state of the atom in reaction to enabled ports.

**Data types and variables**

In BIP2, (data) variables are used to store data. A declaration of a variable is `data` keyword. For example:

```java
data int x
```

declares an integer variable named `x`.

Variables exported with the `export` directive can be used in guards of compound component priorities (see `Compounds`).

• resources needed by tests must be stored in sub-directories of `src/tests/resources/`. Name the sub-directories so that it is easy to match the files to their corresponding test classes.

### 9.4 Common

#### 9.4.1 u.v.bip

This module contains parts that may be shared by every part of the compiler. Currently, it only contains the needed interface and library to parse command line arguments. The `Configurable` interface is used for plug-in after command line has been parsed: arguments are passed to the plug-in so that it can configure itself.

#### 9.4.2 u.v.b.error

The error module is the base of all error handling in the compiler. The main idea behind it is the following:

- an error type has a unique identifier across all compiler: all identifiers are defined in this module. This is a major problem concerning modularity as a plug-in must have its specific errors defined in the base of the compiler.
- error messages are not hardcoded and are shipped as properties. Currently, only an English version is available, but translating the few dozens of message is straightforward.

All errors must inherit from the `GenericError` class. This class defines the most common attributes needed to handle error and display useful error message to the user:

- the error code
- when possible, the location in the BIP source file

The error identifiers are defined as an enumerated type in `ErrorCodeEnum`. The class `ErrorMessage` must be used to get human readable error messages. Its `getMessage()` method takes an error identifier and returns the corresponding error message from the property file used when starting the compiler (by default, it uses the `english-messages.properties` file).

If you need the user to designate a given warning, you should use the helper mapping “`userFriendlyNames`” provided within "`ErrorcodeEnum`". It maps names that the user can easily understand to internal names that maybe too verbose to be user friendly. This is what is used by the "`@SupressWarning`" annotation.

**Important:** Having a pluggable system for error handling is completely possible. It has not been implemented yet for simplicity and because of limited development resources. It may be fixed in future versions, if needed.

#### 9.4.3 u.v.b.exception

This package only contains a single class called `CompilerErrorException`. This exception class is unchecked and must be used if and only if a bug in the compiler has been found. This class is very minimalist and contains members that could be useful to track the origin of the bug.

### 9.5 Middle-end

This module contains currently 2 elements:

- the common part that contains the interface between the middle-ends and the user interfaces: the `Filterable` java interface and the necessary classes/enums for error handling.

12 Chapter 2. The BIP2 Language
Atoms have ports declared with the `port` directive that consists of a type, a name and an optional list of previously declared variables. It is an error if the types of the previously declared variables do not match the type of the corresponding port parameters. Implicit type casting is not permitted. For example, if a previously declared parameter is of type `float`, a port parameter of type `int` is not allowed. In the following code excerpt, three variables named `a`, `b` and `c` are associated with the three parameters of the port with type `Port_t`:

```
port type Port_t(int x, float y, some_type z)
atom type SomeType()
data int a
data float b
data some_type c
port Port_t p(a, b, c)
```

Ports can be exported with the `export` directive and become accessible to other model components. Exported ports can be accessed individually in the component interface (see Figure 2.2) or merged into one port (see Figure 2.3). In the later case, they must accept the same number and types of parameters. The merged port provides access to all variables of the individual ports.

![Figure 2.2: Ports](image1.png)  
**Figure 2.2: Ports**

In BIP2, ports `p`, `q` and `r` are individually exported using the following statement:

```
export port port_t p(x), q(y), r(z)
```

![Figure 2.3: Ports](image2.png)  
**Figure 2.3: Ports**

In BIP2, ports `p`, `q` and `r` are merged and exported as the port `exp`.

```
export as port exp(p, q, r)
```

Port instantiation in the hierarchy is unrolled as described above. All sub-components (e.g., ports, data, connectors, components, etc.) are instantiated during the unroll phase. Components parameter need special handling as these are only assigned after the instantiation phase.

```
instantiate(*Declaration declaration)
```

The actual class and mean of transmission depends on the step failing.
export port post port_t p(x), q(y), r(z) as exp

Petri net

Petri nets implement the behavior of atoms. They consist of places and transitions. Places are used to store the current control location of the atom given by a marking of the places, that is, a boolean function associating true to the marked places. Places are declared in an atom using the keyword places followed by a list of place names, e.g. the following code declares the places named START, SYNC and END:

places START, SYNC, END

Transitions change the current state of an atom and invoke associated actions that may alter the values of atom variables. A transition specifies:

- The set of triggering places that are required to be all marked at the current state for the transition to occur. They are declared using the keyword from. For example, the following code fragment specifies the initial transition of an atom that marks the place START and initializes the variables x and y:

  initial to START do { x=0; y=0; }

- The set of target places that are marked after its execution. They are declared using the keyword to. For example, the following code fragment specifies the internal transition that changes the current state from SYNC to END:

  internal from START to SYNC provided (x!=0) do { x=f(); }

- A boolean condition on values of (local) variables that must be fulfilled at the current state for the transition to occur. This condition, called guard, is declared using the keyword provided. If no expression is provided, the guard places no restrictions on the transition.

  • An optional block of code after the do keyword that is evaluated when the transition occurs.

  A transition of an atom is enabled if:

  • it is enabled by the marking, that is, all its triggering places are marked at the current state and
  • the associated guard evaluates to true or there is no guard associated with the transition.

Important: Notice that in BIP2 we target 1-safe Petri nets where the target places of an enabled transition are never marked. This property for a Petri net of an atom is checked both at compile time and at run time, and leads to an error if violated. Notice that since automata are a sub-case of 1-safe Petri nets, they can be used to define the behavior of atoms. In automata, each transition has at most one triggering place and one target place.

We distinguish three types of transitions:

- The initial transition is responsible for initializing the marked places and atom variables. It is a mandatory transition executed once during the model initialization. It has no triggering places and no associated guard. Moreover, the initial transition can not be observed by other components nor synchronized with their transitions. For example, the following code fragment specifies the initial transition of an atom that marks the place START and initializes the variables x and y:

  initial to START do { x=0; y=0; }

- Internal transitions are invisible to other components and take precedence over other observable transitions. Their execution depends on the current state and associated guards. Internal transitions are declared using the keyword internal, e.g. the following specifies an internal transition enabled in the START place that sets the current state to the SYNC place restricted by an associated guard:

  internal from START to SYNC provided (x!=0) do { x=f(); }

- Transitions labeled by internal port names are visible to other components. A transition labeled by an internal port that is exported can be synchronized with transitions of other components using connectors (see language-connector-label). Such transitions are declared in atoms using the keyword on, e.g. the following specifies a transition labeled by the internal port s, that changes the current state from SYNC to END:

  transitions from START to SYNC on s do { x=f(); }

Parser

The parser is using the antlr tool. You can find many GUI for helping in the development of antlr grammar.

The BIP compiler follows the antlr recommended workflow:

- Bip2.g: a regular grammar is used to read a BIP source code. This pass creates an abstract syntax tree (aka. AST) by using antlr automatic tree building.
- Bip2Walker.g: a second grammar expressed on tree is used to recognize the AST created by the previous pass. Rules embed the necessary java code for building a BIP-EMF model. This model describes only the types found in the parsed BIP code; instances are handled later.

The goal of this split is to have the grammar part as language agnostic as possible: rules do not embed any java code. The file grammar/Bip2.g could be used to build parsers for other languages supported by antlr (e.g. ruby or python).

Important: The previous statement is not 100% true, as we want to plug the compiler’s error management inside the parser to be able to rewrap parser’s errors and display present them to the user in a coherent way. There are few lines of java in the header of the grammar file: these lines can be safely removed if the grammar is to be used for a different target language than java.

The java code generated by antlr from the previous two grammar files is not in the code repository. You need to generate it first. Invoking the ant target gencode-for-eclipse should do the job are generate java code in build/generated-src directory.

Important: When you change one of the grammar files, you must regenerate the code.

Important: You must not use directly the gencode ant target as it’s used for packaging the compiler. The generated code won’t be in the correct location for eclipse development.

Package Loader & Package registry

The package loader is a simple object that uses:

- a classloader to locate BIP files across different directories with the dotted package naming.
- a registry, that is nothing more than a hash table, used to store the BIP packages already loaded.

It is the package loader that takes care of running the parser when a BIP file needs to be parsed.

The loader has a very simple interface, mainly consisting of the method:

- Package getPackage(String package_name) : returns the type model corresponding to the package named after the package_name parameter.

9.3.3 u.v.b.instantiate

The instantiate is responsible for creating an instance model from a set of BIP packages and a root component declaration. Its result is a DAG with instance of BIP packages as nodes. The entry point is the method:

- ComponentInstance instantiate(ComponentDeclaration declaration)

It reads the declaration, search for the corresponding type in the loaded types and returns a ComponentInstance object describing an instance of a component. This call will recursively invoke other instantiate
the variables of the enabled internal ports are accessible from the interface. For historical reasons, (i.e., several maximal internal ports are exported through the same port (i.e. merged export), they are all visible to other components that can interact with any of the internal ports through the interface. Consequently, if an internal port references variables, the values accessible from the interface are the values of the enabled maximal internal ports.

Enabled ports of atoms

When adding a new constraint, important:

• generate java code
• implement the constraints
• versionning generated code

Priorities

Priorities are used during compilation so in this case priority validation is postponed until run time. Inconsistencies in priorities (e.g. different handwritten code that we really need to keep track of) from automatically generated code (thousands of lines of code for the package loader and its package registry). This code must be added in the array of constraints.

Versionning generated code

For more information on how to version generated code and commit your changes with the names of the constraints you’ve modified.

Figure 2.4: Sequence of internal and visible transitions in an atom.
example, if ports $x$ and $z$ are enabled, the associated $u$ and $w$ values are accessible from $\exp$. On the other hand, if only port $y$ is enabled, the value associated with port $\exp$ is $v$. This means that when other components interact with $A$ through port $\exp$, depending on which of the internal ports is enabled, they interact with port $p$ using value $u$, or with port $q$ using value $v$, or with port $r$ using value $w$.

![BIP2 Diagram](image)

Figure 2.5: Example of a port enabled by an atom and the corresponding values of its variable.

### Example

```bip2
atom type MyAtom(int P)
data int x
export data int y

port Port_t\_t\_r(x), s(y)

places START, SYNC, END
initial to START\_t\_r do { x=P; y=0; }
internal from \_t\_r to SYNC provided {x!=0} do { y=f(x); }
on \_t\_r from START to SYNC\_r do { y=x; }
on a from SYNC to END\_r

The above block of BIP2 code gives an example of atom type `MyAtom` that accepts one integer parameter $P$, and consists of two integer variables $x$ and the exported variable $y$ and two exported ports $r$ and $s$. Three places, `START`, `SYNC`, `END`, are the states of the automaton that defines the behavior of the atomic component. An initial transition leads to `START`, an internal transition changes the state from `START` to `SYNC`, an other transition triggered by $r$ does the same and finally a transition triggered by $a$ modifies the state from `SYNC` to `END`. A graphical representation of `MyAtom` is provided below.

Since internal transitions have higher priority than port transitions, the transition of port $r$ is executed only if the guard of the internal transition does not hold, i.e. the value of variable $x$ is zero.

### Syntax

- accepts annotations

```bip2
atom_type_definition :=
  'atom type' atom_type_name '(' [ data_parameter (',', data_parameter)* ] ')'  
  | ['export'] 'data' data_type  
  | data_declaration_name (';', data_declaration_name)*
```
2.2. Connectors

Connectors are stateless entities that enable interactions among a set of components via their interface ports. Interactions defined by a connector are strong synchronizations (i.e., a rendez-vous) of a subset of the connected components. Interactions may also include data that are transferred between the components. A connector is hierarchical if it connects ports exported by other connectors.

Connected ports

A connector type accepts a list of typed ports that correspond to the ports of the entities it connects (components or other connectors). Connectors (i.e., instances of connector types) bind these parameters to actual ports of the same type.

Important: Components or connectors must be connected at most once in a connector, that is, a component or a connector must not be reachable from different connected ports.

Data variables

Connector types can define variables that are used for storing intermediate results of computations performed in transfer functions associated with interactions. The temporary stored value is accessible only during the associated interaction. The syntax is shown in the following example where we declare an integer variable named tmp:

```
import

tmp : Integer;
```

9.3. Front-end

9.3.1. u.v.b.metamodel

This module defines the BIP2 meta-model used by all parts of the compiler, as the meta-model is the intermediate representation of BIP models. It contains:

• the meta-model itself, as an .ecore file
• the constraints on the models of this meta-model

The `bip2.ecore` file is located in the `model/` sub-directory. This is the file you need to use with tools dealing with EMF models. It comes with 2 other files:

• `bip2.ecoreдиаг`: it is tied to the `ecore` and allows the graphical editing of the meta-model with EMF editor. Opening this file and editing the displayed model will modify automatically the `ecore` accordingly.

• `bip2.genmodel`: this file is used by the EMF code generator. In BIP, we use only the Java code generation mechanism.

The regular work-flow when touching the meta-model is given below:
data int tmp

Exported port

A connector may export a single port that can be connected to other connector instances and form hierarchical connectors, or it can be exported in the interface of compound components (see Compound). A connector is top-level if the exported port is not connected directly to another connector (i.e., it can be connected to other connectors only at upper levels after being exported by the containing compound), or if it has no exported port. An exported port named \texttt{exp} of type \texttt{port_t}, referencing a variable \texttt{tmp}, is declared in a connector type as follows:

\begin{verbatim}
export port post_t \texttt{exp(tmp)}
\end{verbatim}

Defined interactions

Formally, an interaction of a connector type is a subset of its ports. A connector type explicitly defines a set of permitted interactions regardless of the status of the connected ports. The interactions are defined in terms of expressions involving port names, according to the following grammar:

\begin{verbatim}
connector_port_expression :=
  ( sub_expression )
sub_expression :=
  ( port_name : ( '(' connector_port_expression ')' ) ) | ''
\end{verbatim}

That is, an expression is a list of either port names or nested expressions (expressions enclosed into parenthesis) that can be optionally quoted. Quoted port names or nested expressions are called triggers, whereas unquoted ones are synchrons.

An expression of the form \( p \), where \( p \) is a port name, defines a single interaction \( \langle p \rangle \). Interactions defined by an expression of the form \( e^1, e^2, \ldots, e^N \) are computed recursively from the interactions defined by sub-expressions \( e^1, e^2, \ldots, e^N \), as explained as follows. An interaction is defined by \( e^1, e^2, \ldots, e^N \) if both following rules apply:

- it can be written as a union of interactions defined by sub-expressions \( e^1, e^2, \ldots, e^N \)
- it contains (at least) an interaction defined by a trigger sub-expression, or for each sub-expression \( e^i \), \( I=0,\ldots,N \), it contains an interaction defined by \( e^i \).

In the following example we define one trigger sub-expression \( p, q \), and two synchrons ports \( r \) and \( s \):

\begin{verbatim}
define \( p, q \) r s
\end{verbatim}

Interactions permitted by such an expression are the ones containing (at least) both ports \( p \) and \( q \), i.e. \( \langle p, q \rangle, \langle p, q, r \rangle, \langle p, q, s \rangle \), and \( \langle p, q, r, s \rangle \).

Guards and transfer functions

The set of defined interactions in a connector type can be further restricted by guards. Guards evaluate a boolean expression that refers to variables of the ports involved in an interaction. The associated interaction is enabled only if the guard evaluates to true.

Transfer functions are used for exchanging data between the components that participate in an interaction. They consist of two instruction groups, the up and the down group.

The up instructions compute the values of the variables referenced by exported ports. Also, intermediate values used in computations in the down section may be temporary stored in the connector’s variables. In the following example

\begin{verbatim}
9.2.1 ivy

Ivy is used to define the dependencies between all modules and in conjunction with ant or eclipse, for the correct building of the compiler. Each module contains the following files:

\begin{verbatim}
ivy.xml
\end{verbatim}

This file simply contains information on the dependencies of the module. When a module depends on another module, ivy automatically computes the transitive dependencies. When a module depends on an external library (eg. a jar file), it simply declares this dependency and ivy will take care of not uselessly duplicating this jar file because of transitive dependencies.

The following excerpt from \texttt{ujf.verimag.bip} module in Common shows the 2 types of dependencies:

\begin{verbatim}
<dependency name="jopt-simple" rev="3.2">
<artifact name="jopt-simple" type="jar" url="file:/path/to/dependencies/jopt-simple-3.2.jar" />
</dependency>
<dependency name="ujf.verimag.bip.error" rev="latest.integration"></dependency>
\end{verbatim}

It defines 2 dependencies:

- the first one, named \texttt{jopt-simple}, at version 3.2. This dependencies is \textit{direct} as we also provide the corresponding \textit{artifact} (a path to the jar file).
- the second one, named \texttt{ujf.verimag.bip.error}. As there is no more information, ivy will have to find the provider for this dependency (in this case, the \texttt{ujf.verimag.bip.error} module).

A single dependencies can have several artifacts, as is the case of the EMF in \texttt{ujf.verimag.bip.matomodal} module:

\begin{verbatim}
<dependency name="org.eclipse.emf.ecore" rev="2.7.0">
  <artifact name="org.eclipse.emf.ecore" type="jar" url="file:/path/to/dependencies/org.eclipse.emf.ecore_2.7.0.v20110331-2022.jar" />
  <artifact name="org.eclipse.emf.ecore.ecore" type="jar" url="file:/path/to/dependencies/org.eclipse.emf.ecore.ecore_2.7.0.v20110331-2022.jar" />
  <artifact name="org.eclipse.emf.ecore.ecore.ecore" type="jar" url="file:/path/to/dependencies/org.eclipse.emf.ecore.ecore.ecore.ecore.ecore_2.7.0.v20110331-2022.jar" />
</dependency>
\end{verbatim}

The full documentation on ivy can be found at \url{http://ant.apache.org/ivy/}

\begin{verbatim}
build.xml
\end{verbatim}

This file is used by \texttt{ant} to schedule the build. This includes the actual compilation of source files (accelero templates, antlr grammar, java code, ...) and the use of ivy to resolve each module’s dependencies.

Module with only java code in the src/main/java directory have a 3 liner as \texttt{build.xml}:

\begin{verbatim}
<project name="ujf.verimag.bip.FOO" default="compile">
  <property file="build.properties" />
  <import file="${distribution.dir}/common.xml" />
</project>
\end{verbatim}

9.2. Generalities
9.1.3 Front-end

setup a correct development environment.

Before describing every internal part of the compiler, we need to describe how the build system works and how to

9.2 Generalities

2.2. Quick overview of the language 19

Example

Figure 2.6: Connector type example

2.2. Quick overview of the language

The middle-end only contains the needed mechanics so that... system works and how to

Example

Figure 2.6: Connector type example

2.2. Quick overview of the language

The middle-end only contains the needed mechanics so that... system works and how to
Enabled interactions and ports exported by connectors

The set of ports defined by an interaction is restricted at run time based on the status of the involved ports and the evaluated guards. Let us consider an example of interaction involving ports p, q, and r:

\[
\text{define } p' \quad q \quad r
\]

Based on the definition above, the permitted interactions are \( (\text{'p'}, \text{'p'}, \text{'q'}, \text{'p'}, \text{'r'} ) \) and \( (\text{'p'}, \text{'q'}, \text{'r'} ) \). To determine which of the combinations are valid in a model execution, we first remove all combinations that contain a disabled port and then we evaluate the associated guards to further restrict the possible combinations.

An exported port of a connector is enabled if there is at least one enabled interaction. Notice that the value visible at the interface through the exported port is derived by the set of values of ports participating in an interaction. The values accessible from an enabled interaction are in turn computed by the instructions of the \( \text{up} \) transfer function.

The notions of enabled interactions and corresponding values of the variables of exported ports of connectors are illustrated by the following example. Consider an instance \( C \) of the connector type \( \text{ConnectT} \) presented above. Assume that ports \( p, q, r \) are enabled, and that variable \( x \) of port \( p \) has three possible values \( u1, u2, u3 \), variable \( y \) of port \( q \) has three possible values \( v1, v2, v3 \), and variable \( z \) of port \( r \) has only a single value \( w \). Then, interactions \( (\text{'p'}, \text{'p'}, \text{'q'}, \text{'p'}, \text{'r'} ) \) and \( (\text{'p'}, \text{'q'}, \text{'r'} ) \) are enabled. Moreover, there are 24 possible values for variable \( \text{tmp} \) of the exported port \( \text{exp} \), corresponding to the application of \( \text{up} \) to all combinations of values for the 4 enabled interactions:

- The values corresponding to interaction \( p, q \) are the values of \( x \), that is, \( u1, u2, u3 \).
- The values corresponding to interaction \( p, q, r \) are \( oIJ = \max(u1, v2, v3) \), such that \( I=1,2,3 \) and \( J=1,2,3 \).
- The values corresponding to interaction \( p, r \) are \( oI = \max(uI, w) \), such that \( I=1,2,3 \).
- The values corresponding to interaction \( p, q, r \) are \( oIJ = \max(u1, v2, v3, w) \), such that \( I=1,2,3 \) and \( J=1,2,3 \).

![Figure 2.7: Enabled interaction of a connector and the corresponding values of variable \( \text{tmp} \).](image)

Syntax

- accepts annotations

```latex
definition =
  'connector_type' connector_type_name '(' 'port_parameter (', 'port_parameter )' )* (
  'data' data_type data_declaration_name (', ' data_declaration_name )* ) |
  'export port' port_type
  'export param' ( 'data_param_declaration (', ' data_param_declaration )' ) |
  'define' connector_port_expression
  connector_interaction*
```

9.1 Compiler design

Goals:

- users/developers usually write different code generator: adding a new code generator should be as easy as possible
- users/developers usually enrich the input language for driving the code generator. Avoid the burden of changing the core grammar as this is very often overkill.

Big picture:

- front-end: \( \text{any to BIP-EMF} \) transformation. Takes any source code in a given language and translate it to \( \text{BIP-EMF} \)
- middle-end: \( \text{BIP-EMF to BIP-EMF} \). Apply operations on a \( \text{BIP-EMF} \) input (operations can be read and/or write).
- back-end: \( \text{BIP-EMF to any} \). Generates source code in a given language from a \( \text{BIP-EMF} \) input.

The current BIP compiler is developed with eclipse, but this is not a hard requirement. Not using eclipse can be a bit hard because of the compiler use of some eclipse technologies (in particular, \( \text{EMF} \)). The compiler is composed of more than 10 different modules, each module being a single eclipse project. The layout must be the following one:

```
  \text{Common} |
  \text{Frontend} |
  \text{Middleend} |
  \text{Backend} |
```

We give here a very brief description of each module. Full description is given in the following sections.
connector_interaction :=
'on' (port_name)+
['provided' '(' connector_guard ')']
['up' '{' (statement ';')+ '}']
['down' '{' (statement ';')+ '}']

"connector_port_expression := (port_name | '(' connector_port_expression ')')+
(a connector interaction must have at least one of 'up', 'down' or 'provided')"

2.2.9 Compounds
Compounds are composite components constructed by atomic components and other compound components. Just like atomic components, compounds provide a set of ports at the interface level. In this sense, components are used in the same way regardless of their structure (compounds or atomic).

A compound type defines the following.
• a set of components, either atomic or compound, declared with the keyword component.
• a set of connectors declared with the keyword connector that connect the contained components.
• a set of priority rules declared with the keyword priority.
• a set of exported ports that define the interface of the compound declared with the keyword export.

Notice that a compound component can export ports of contained components as well as ports of connectors.

Priorities
Priorities are used to favor the execution of a subset of enabled interactions called the maximal interactions (see below for a definition of maximal interactions). They can be used to resolve conflict between interactions or to express particular scheduling policies.

Priorities of a compound form a partial order relationship that corresponds to the transitive closure of the defined priority rules. One set of priority rules is automatically derived based on the maximal progress principle, i.e. interactions that involve more connectors have higher priority.

User-defined priority rules are of the form

\[ I < J \]

where \( I \) and \( J \) are interactions of connectors expressed in one of the following forms:

• \( C:A_1.p_1,A_2.p_2,...,A_N.p_N \)
  where \( C \) is a connector and \( A_1.p_1,A_2.p_2,...,A_N.p_N \) is a subset of the connected ports that corresponds to a defined interaction of \( C \).

• \( C:* \)
  where \( C \) is a connector represents all the defined interactions of \( C \).

• \(*:* \)
  represents all the defined interactions for all connectors.

Important: User-defined priority rules can only involve interactions of top-level connectors.

The use of \(*\) in priority rules is a shortcut for sets of rules. Notice that \(*:*\) cannot be used for both sides of a priority rule, (e.g. \(*:* < *:*\) is not allowed). The use of \(*:*\) in one side of a priority rule is a shortcut for all interactions defined in all connectors except those involved in the other side of the rule.
User-defined priority rules may include guards declared with the provided keyword. A rule is enabled only if its guard evaluates to true. In the following code excerpt we show a priority rule named myPrio that is enabled only if the values of the x and y variables of the atomic components A and B are not the same:

```
compound type Compound_T()
  component Atom_T A()
  component Atom_T B()
  connector RDV C(A.p,B.p)
  connector RDV D(A.q,B.q)
  priority myPrio provided (A.x != B.x) C:A.p,B.p < D:A.q,B.q
end
```

Important: Since priorities define a partial order relationship between interactions, priority rules enabled at a state of a compound must not form a cycle.

An enabled interaction I of a connector C has lower priority than an enabled interaction J of a connector D if D is reachable from I in the lattice of the defined priority rules, that is, if C:I < D:I is an enabled rule or if there exists interactions C1:I1, ..., CN:IN such that rules C:I < C1:I1, C1:I1 < C2:I2, ..., CN-1:IN-1 < CN:IN, CN:IN < D:I are enabled. An interaction is maximal if it has the highest priority among the enabled interactions.

### Exported ports and variables

Compound types export ports and variables in a similar fashion with atoms. The following statement makes the x variable accessible from the interface of the A component and renames it to y:

```
export A.x as y
```

Ports of components and connectors can be exported individually or through a single port using a merged export, in the same way as atoms. To determine if a port of a compound component is enabled we check if the underlying port (component or connector port) is enabled. If a port of a component is enabled and exported, then the corresponding port at the interface is enabled. If a (maximal) interaction is enabled in a connector that exports its port to the interface of a compound, then the interface port is enabled. Moreover, values visible at the interface are the values corresponding to all its maximal interactions. As for atoms, for merged exported ports the union of the values is visible at the interface.

```
compound type Compound_t()
  component Atom_t A(), B()
  connector Connector_t C1(A.p, B.p)
  connector Connector_t C2(A.q, B.q)
  export C1.exp, A.r, B.r as s
  priority myPrio C1:A.p,B.p < C2:*
end
```

In the above example, the port s of an instance of the compound type Compound_t is enabled if the connector C1 has a maximal interaction (i.e. if no interaction is enabled by C2), or if port r of A is enabled, or if port r of B is enabled. Moreover, if these ports have variables, the values visible from s are the union of the values corresponding to the maximal interactions of C1 and the values visible from ports r of A and B.

### Example

```
compound type CompT1 K1()
```
The above example shows the syntax for defining a compound type Compound_t that consists of:

- the components K1, K2, and K3
- the connectors C, D, and E, such that C and D are connected and form a hierarchical connector
- the exported ports xp of C and F and the exported port t of K3
- the exported variable x of K3.

A graphical representation of the compound type is provided below. Notice that all the enabled interactions of connector C are visible from connector D through the port xp of C, e.g. if p and p,q are enabled, there are both visible from D. Since priorities are applied when exporting ports to the interface of a compound, only maximal interactions of C are visible from the interface port u though xp, e.g. if interaction p and p,q are enabled, only p,q is visible from u due to the default priority rule of maximal progress: p < p,q. Notice also that a port can be connected to several connectors (e.g. port q of K2), or can be exported and connected to connector(s) (e.g. ports xp of C and t of K3).
2.3 Execution sequences

A BIP2 model is equivalent to a labeled transition system (LTS) that defines all the allowed execution sequences. The model state is stored in the state of atomic components represented by variable values and the marking of the Petri nets. An execution sequence is a sequence of transitions or interactions that modify the global state. The transitions and interactions that are available in a certain state are defined as follows:

- A transition of an atom A is executed from a state if it is enabled, is maximal and is not labeled by an exported internal port.
- An interaction of a connector C is executed if it is enabled, is maximal and connector C does not export a port.

In a given state, only the non exported maximal transitions and interactions are allowed. During their execution, non maximal exported transitions or interactions are executed according to the hierarchy of connectors in the model.

The execution of an enabled transition modifies the current state as follows:

- marking of Petri nets are modified according to the triggering and target places of transitions, i.e. marks are removed from triggering places and are set in target places
- variables are modified by the code associated with the transition.

**Important:** If a place is both a triggering and a target place for a transition, its mark remains unchanged.

An interaction 'p1,p2,...,pN' of a connector C, considering a particular combination of values for its ports, modifies the model state as follows:

First, the instructions associated with the down transfer function are performed for the values of the involved ports p1,...,pN. Then, the state is modified according to the execution of ports p1,...,pN:

- The execution of an atom port is equivalent to the corresponding transition.
- The execution of a compound port corresponds to the execution of the corresponding port.
- The execution of a connector port corresponds to the execution of the corresponding interaction.

```plaintext
export port: port_reference (',', port_reference)* as exported_name

export data: data_reference as exported_name
```
Important: The execution of an interaction corresponds to the execution of at most one transition of each atom of the model. Since atoms have disjoint sets of variables and places, the state of the model resulting from the execution of an interaction is independent from the order of execution of the involved atoms.

2.3. Execution sequences

comp_type_data_params
: native_data_type_param (COMMA native_data_type_param)*

atom_priority_guard
: LPAREN logical_or_expression RPAREN

port_name_wildcard
: simple_name | MULT_OP

atom_priority_declaration
: PRIORITY simple_name port_name_wildcard LT_OP port_name_wildcard (PROVIDED atom_priority_guard)?

atom_type_definition
: ATOM TYPE simple_name LPAREN (comp_type_data_params)? RPAREN (multi_data_declaration_with_modifiers)* (multi_port_declaration_with_modifiers)*

places_declaration
initial_transition
transition+
atom_priority_declaration*
END

fragment_component_declaration
: simple_name LPAREN (logical_or_expression (COMMA logical_or_expression)*)? RPAREN

multi_component_declaration
: annotation* COMPONENT fully_qualified_name fragment_component_declaration (COMMA fragment_component_declaration)*

fragment_connector_declaration
: simple_name LPAREN fully_qualified_name (COMMA fully_qualified_name)* RPAREN

multi_connector_declaration
: CONNECTOR fully_qualified_name fragment_connector_declaration (COMMA fragment_connector_declaration)*

export_inner_port
: annotation* EXPORT PORT fully_qualified_name (COMMA fully_qualified_name)* AS simple_name

export_inner_data
: annotation* EXPORT DATA fully_qualified_name AS simple_name

96 Chapter 8. BIP 2 Grammar
LPAREN data_type_name RPAREN

data_types_params
  : data_type_name (COMMA data_type_name)*

annotated_const_data_declaration
  : annotation* 
    CONST DATA native_data_type_name simple_name ASSIGN_OP logical_or_expression

places_declaration
  : PLACE simple_name (COMMA simple_name)*

transition_action
  : LBRACE! ((statement SEMICOL!) | if_then_else_expression) * RBRACE!

transition_guard
  : LPAREN logical_or_expression RPAREN

transition
  : annotation* 
    (ON simple_name | INTERNAL) 
    FROM simple_name (COMMA simple_name)* 
    TO simple_name (COMMA simple_name)* 
    (PROVIDED transition_guard)? 
    (DO transition_action)?

compound_interaction
  : simple_name COLON (fully_qualified_name (COMMA fully_qualified_name|MULT_OP)

compound_interaction_wildcard
  : compound_interaction | MULT_OP COLON MULT_OP;

compound_priority_guard
  : LPAREN logical_or_expression RPAREN

compound_priority_declaration
  : PRIORITY simple_name 
    compound_interaction_wildcard LT_OP compound_interaction_wildcard 
    (PROVIDED compound_priority_guard)?

initial_transition
  : INITIAL TO simple_name (COMMA simple_name)* (DO transition_action)?
CHAPTER
THREE
COMPILER AND ENGINES PRESENTATION

3.1 The compiler

The compiler consists of three parts that will be presented in more details in the following sections:

• the **front-end**: it interacts with the user of the compiler. It reads user input and transforms it in a form suitable for the following process (ie. internal representation).

• the **middle-end**: applies operations on the internal representation (eg. optimizations, architectural transformations, ...). One such operation is contained into a small block in the compiler that we will call later filter.

• the **back-end**: produces the final result from the internal representation. Usually in the form of a source code in a programming language (eg. C++). Several back-ends can be used at once.

![Figure 3.1: Overview of Compiler design](image)

A typical compilation consists of the following steps:

• first, the front-end executes and creates a BIP-EMF model.

• then the filters in the middle-end are executed in turn. The result is a possibly modified BIP-EMF model.

• finally, all back-ends are executed in turn. Their results are the compilation results.

3.1.1 The Front-End

This part is responsible for reading user input (ie. BIP source code & command line argument) and transforming it into an intermediate representation that will be used throughout the other parts of the compiler. The current front-end consists of the following components:

- The **tokenization** module takes the input string and transforms it into a sequence of tokens, each of which represents a syntactic unit.

- The **parser** module uses the grammar rules to build an abstract syntax tree (AST) from the tokens produced by the tokenizer.

- The **intermediate representation** module transforms the AST into a more compact and efficient representation that can be used by subsequent modules.

The compiler consists of these parts that will be presented in more details in the following sections.
end contains a parser for the BIP language and a BIP meta-model that describes the intermediate representation. An instance of a BIP model represented in the BIP meta-model is called a BIP-EMF model (because it is a BIP model expressed using the Eclipse Modeling Framework (EMF) technology) in the following text. For more details on the internals, see Front-end.

**Type model versus Instance model**

The BIP language only deals with types. There is no support for running entities, even if the final result should be a running system. This missing information is usually filled by specifying a root component at compile time. The compiler (i.e. the front-end) is then able to build both a type model (i.e. a representation of the BIP source code given as input) and an instance model that represents the system you want to run. The distinction between the two can be subtle, especially when the concept of declaration is mixed in between:

- a component type describes the shape of an instance of that type
- a component declaration instructs the creation of an instance of a component type
- a component instance is a running entity

These notions are similar to class-instance/object declaration that can be found in object oriented language. For example, in Java:

- a component type = a class:
  ```java
  public class MyClass { ... }
  ```
- a component (instance of a component type) = an object (instance of a class):
  ```java
  new MyClass();
  ```
- a component declaration = an object declaration:
  ```java
  MyClass m;
  ```

Beware that a component declaration can trigger the creation of more than one instance. A component declaration usually is not a discriminant component identifier within the whole system.

### 3.1.2 The Middle-End

The middle-end hosts all the BIP to BIP transformations. It acts on the BIP-EMF model by means of operations (but are not limited to):

- architectural modifications (e.g. flattening, component injection, ...)
- petri net simplifications
- dead code removal
- data collection

The compiler currently does not have any such operation: the middle-end is empty. See Middle-end for more details.

### 3.1.3 The Back-End

The back-end gets the BIP-EMF model and is only allowed to read it and produce something, most probably some source code in another language (e.g. C, C++, Aseba, ...) or even in BIP. Currently, the main back-end used is the C++ back-end that produces C++ code suitable for standard engine (see Installing & using available engines for the definition of a standard engine).
Several back-ends can be used at once; for example, you may need to get a BIP version of your input after some optimizations have been applied along with its corresponding C++ version. Compiler design forbids back-ends to interact (when there are several back-ends to execute, the compiler does not specify in which order they will be run or if the executions will be in parallel or not).

### 3.2 The engines

An engine takes some representation of a BIP model and computes corresponding execution sequences according to the BIP semantics. Usually, the representation used is a C++ software that is linked against the engine's runtime to create an executable software. Typically, engines target one or more of the following main goals:

- **Execution**: the model corresponds to the computation of a single execution sequence that is intended to be executed on the target platform. In this case, the engine realizes the connection between the model and the platform in order to ensure a correct behavior of the execution with respect to timing and input/output data (through sensors/actuators).

- **Simulation**: the model corresponds to the computation of a single execution sequence that is intended to be executed on the host machine for simulation purpose, that is, time is interpreted in a logical way.

- **Exploration**: the model corresponds to the computation of several execution sequences corresponding to multiple simulations of the model. Model-checking of the model requires a full coverage of the execution sequences defined by the application of the semantics, but a partial coverage can be sufficient for validation or statistical model-checking.

### 3.3 The interactions between the engines and the compiler

Typically, a back-end generates source code from a BIP model. This source code is then associated with a runtime, called an engine, that is responsible for the correct execution of the BIP model with respect to the BIP semantics. The generated source code could be seen as yet another representation of the BIP model (with nothing added to the information contained in the BIP source code) suitable for a given engine (that implements the semantics of the language).
BIP 2 GRAMMAR

The full grammar is given with antlr syntax. The Java code & some header have been omitted for readability.

```antlr
grammar Bip2;

CT_INT : 'int';
CT_BOOL : 'bool';
CT_FLOAT: 'float';
CT_STRING: 'string';

TRUE : 'true';
FALSE : 'false';
REFINE : 'refine';
EXTERN : 'extern';
EXPORT : 'export';
FUNCTION : 'function';
OPERATOR : 'operator';
DEFINE : 'define';
DATA : 'data';
PATTERN : 'pattern';
END : 'end';
USE : 'use';
AS : 'as';
ATOM : 'atom';
COMPOUND : 'compound';
COMPONENT : 'component';
ON : 'on';
INTERNAL : 'internal';
DO : 'do';
PROVIDED : 'provided';
INITIAL : 'initial';
PLACE : 'place';
FROM : 'from';
TO : 'to';
PRIORITY : 'priority';
CONNECTOR : 'connector';
UP_ACTION : 'up';
DOWN_ACTION : 'down';
PORT : 'port';
TYPE : 'type';
CONST : 'const';
LPAREN : '(';
RPAREN : ')';
LBRACE : '{';
```

CHAPTER
FOUR
INSTALLING & USING THE BIP COMPILER

4.1 Requirements

BIP compiler is currently only tested on GNU/Linux systems. It is known to work correctly on Mac OS X, and probably other Unices, but we do not support them currently.

Before installing the compiler, you must install:

• Java VM, version 6 (or above) for the core compiler. We have mainly used OpenJDK.

Tip:

On GNU/Debian Linux and its derivative (e.g., Ubuntu), you can install this dependency with:

$ apt-get install openjdk-6-jre

Warning:

These instructions cover the installation of the compiler. The common usage involves the generation of C++ code and needs the use of an engine. The quick installation contains the engines. If you are not using the quick installation procedure, see Installing & using available engines for engine installation instructions.

4.2 Downloading & installing

4.2.1 Getting latest version

Go to the download page for the BIP tools. You are offered two solutions to install the BIP compiler and engines:

• the first is easier and quicker but may not fit all systems. Compiler and engines are packaged in the same archive and setup scripts are provided.

• separate archives for compiler & engines are also provided. The installation of the compiler using these archives is explained in a second step.

Quick installation using self-contained archive

For using the quick installation, you need to download the bip-full_<ARCH>.tar.gz archive. Replace <ARCH> with your own architecture (e.g., i686). Then simply follow the following steps:

• create a directory where everything will be installed:

$ mkdir bip2

• extract the archive:

31

BIP2 Documentation, Release 2015.04 (RC7)

my_print("Hello World", somedata);

end

compound type HelloCompound()

component HelloAtom c1()

end

end

And the two externals files.

HelloPackage.cpp:

#include <iostream>

void my_print(const char *message, int &adata){

std::cout << "Someone says: " << message << " with data= " << adata << std::endl;

}

void my_modify(int &adata){

adata = 999;

}

and

HelloPackage.hpp:

void my_print(const char *message, int &adata);

void my_modify(int &adata);

You can ask GDB to add a breakpoint on any transaction guard/action by giving the file+line number, as you would with regular C/C++ debugging (you can use file completion):

(gdb) b HelloPackage.bip:16

Breakpoint 1 at 0x805f649:

qfile /path/to/debug_bip_level/HelloPackage.bip, line 16. (4 locations)

(gdb) r

Starting program: /path/to/debug_bip_level/build/system

Breakpoint 1, AT_HelloAtom::initialize (this=0x8082de0) at

/path/to/debug_bip_level/HelloPackage.bip:16

Current language: auto

The current source language is "auto; currently c++".

GDB displays correctly the position within BIP source code:

| 12 on p from START to S do {
| 13 my_modify(somedata);
| 14 } on p from S to END do {
| 16 my_print("Hello World", somedata);
| 17 } end
| 19
| 20 compound type HelloCompound()
| 21 component HelloAtom c1()
$ cd bip2 ; tar xzvf /path/to/bip-full_i686.tar.gz
bip-full/
bip-full/BIP-reference-engine-2012.04_Linux-i686/
bip-full/BIP-reference-engine-2012.04_Linux-i686/include/
...
• setup the environment (works only in a bash shell):
  $ cd bip-full
  $ source ./setup.sh
Environment configured for engine: reference-engine
By default, setup.sh configure the installation for the reference engine. If you wish, you can also select the optimized engine or the multithread engine by passing respectively optimized-engine or multithread-engine to setup.sh, e.g. to select the optimized engine use:
  $ cd bip-full
  $ source ./setup.sh optimized-engine
Environment configured for engine: optimized-engine

Using separate archives for compiler
The archive name should resemble bipc_2012.01.tar.gz, the version number being dependent of the latest version at the moment you are downloading it.
The compiler is a self-contained archive that you need to extract in a dedicated directory, for example /home/a_user/local/bip2:
  $ mkdir /home/a_user/local/bip2
  $ cd /home/a_user/local/bip2
  $ tar xzvf /path/to/the/bipc_2012.01.tar.gz
bipc-2012.01/
bipc-2012.01/lib/
bipc-2012.01/lib/org.eclipse.acceleo.common_3.2.0.v20111027-0537.jar
bipc-2012.01/lib/lpg.runtime.java_2.0.17.v201004271640.jar
...
bipc-2012.01/bin/
bipc-2012.01/bin/bipc.sh
...
Then, you need to add /home/a_user/local/bip2/bipc-2012.01/bin to your PATH environment variable.
In bash:
  $ export PATH=$PATH:/home/a_user/local/bip2/bipc-2012.01/bin
In tcsh:
  $ setenv PATH $PATH:/home/a_user/local/bip2/bipc-2012.01/bin

4.2.2 Quick tour of installation
After installation, you should get something similar to the following setup:

7.6.8 Debugging at the BIP level
By using the gencpp-enable-bip-debug, it is possible to use the GDB on the BIP source code and not only on the generated C++ code.
Let’s reuse previous example that makes use of external code and modify atom data:

```cpp
@cpp(src="ext-cpp/HelloPackage.cpp",include="HelloPackage.hpp")
package HelloPackage
extern function my_modify(int)
extern function my_print(string, int)
port type HelloPort_t()
atom type HelloAtom()
data int somedata
place START, S, END
initial to START do {
  somedata = 0;
}
on p from START to S do {
  my_modify(somedata);
}
on p from S to END do {
```

```
4.3.4.1 Adding support for BIP model correctness

The compiler always checks if a given input is valid with respect to the language (e.g. syntax is correct, presence of cycles in priorities, correct data flow in up/down of connectors). These checks are applied to both models (type & instance). The compiler may emit two kinds of messages:

- **WARNING**: a potential error has been detected, but it may be a false positive because of runtime dependency. Example of such warning is a cycle in priorities with at least one guarded priority: if the guard is false when all rules apply, then there is no cycle. These messages are preceded by `WARNING` by the compiler.

- **ERROR**: an error has been found and the compiler stops as soon as possible. The input is not correct. A cycle in priority rules and writing to bound port's of a connector during the up phase are examples of such errors. These messages are preceded by `SEVERE` by the compiler.

**Tip:** The compiler can treat warnings as errors and stop compilation when `--Werr` is used (very similar to regular C/C++ compiler behavior regarding `-Werr`).

Sample output with a fatal error (the root declaration references a type that the compiler could not find):

```
$ bipc.sh -p ASamplePackage -d "ThisTypeDoesNotExists()" -I .
[SEVERE] Type not found : ThisTypeDoesNotExists
```

Sample output with a warning (there may be more than one internal transition from the same state, depending on the guards):

```
$ bipc.sh -p ASamplePackage -d "SomeCompoundType()" -I .
[WARNING] In ASamplePackage.bip:
Transition from this state triggered by the same port (or internal) already exists:
19: on tic from S1 to S3 do { c = c + 1; tosend = tosend + 1; start = 1;}
20: internal from S3 to S2 provided (c <= 10)
----------^ 21: internal from S3 to S1 provided (c > 10)
22: on toc from S2 to S1 provided (c < 10)
```

### 7.6.7 Adding serialization support for custom type

Serialization support is useful as the data values are displayed in execution trace. In order to support serialization for custom types, you need to provide a function for the `<<` operator:

```
ostream& operator<<(ostream &o, const CustomType &value);
```

All the work for adding the support takes place in the external C++ code; the BIP source file is the same as the previous example.

We provide below the modified version of the external code.

**HelloPackage.hpp**

```
#ifndef HP_HPP
#define HP_HPP
#include <iostream>
struct __my_custom_type;
struct __my_custom_type {
    int x, y;
    friend std::ostream& operator<<(std::ostream &o, const struct __my_custom_type &value);
};
typedef struct __my_custom_type my_custom_type;
void print_data(int id, my_custom_type &adata);
void init_data(int id, my_custom_type &adata);
#endif
```

**HelloPackage.cpp**

```
#include "HelloPackage.hpp"
void print_data(int id, my_custom_type &adata){
    std::cout << "Data for: " << id << " = " << adata.x << ", " << adata.y << std::endl;
}
void init_data(int id, my_custom_type &adata){
    adata.x = id * 2;
    adata.y = id * 8;
}
std::ostream& operator<<(std::ostream &o, const struct __my_custom_type &value){
    o << 
```
When you run the compiler, you need to provide at least the following parameters:

- a package name to compile: `-p` followed by the package name. The package name must match the file name that contains it (e.g., package `Sample` must be stored in a file named `Sample.bip`).
- one or more package search directories. This list of directories is used by the compiler to look for the package to compile (and the potential other packages that are needed because of dependencies): `-I` followed by a directory. Use the parameter several times to use multiple directories. The compiler will use the first correct match when searching (order is important).

By using only these two parameters, the compiler will load the types contained in the package (and its dependencies) and check them for validity. Nothing is produced by default.

You can also create an instance model along with the type model by giving the compiler a component declaration using a type from the loaded package:

- `-d` followed by a declaration (e.g., `-p ACompound(1,2)`). Beware that it may be required to enclose the declaration by `""` in order to protect it from being interpreted by your shell.

Example execution of the compiler:

```
$ bipc.sh -p SamplePackage -I /home/a_user/my_bip_lib/ -d "MyType()"
```

### 4.3.1 Silencing warnings

Some warnings can be silenced. This is useful when you are 100% sure that the warning is not a problem in your specific case. You must never silence a warning because you don’t understand its presence!

To suppress a warning, you need to attach a `@SuppressWarning` annotation on the element that triggers the warning along with the type of warnings you want to silence. For example, in case of possible non-determinism in a petrinet:

```plaintext
on work from a to a provided (x == 1) do { Max = 0; }
on work from a to a provided (x > 1) do { Max = 0; }
```

The compiler will output:

```
[WARNING] In bla.bip:
Transition from this state triggered by the same port (or internal) already exists: 
  108:  on work from a to a provided (x == 1) do { Max = 0; }
  -----------
  110:  on work from a to a provided (x > 1) do { Max = 0; }
  111: 
```

You can silence this warning by adding annotations:

```plaintext
@SuppressWarning(nondeterminism) on work from a to a provided (x == 1) do { Max = 0; }
@SuppressWarning(nondeterminism) on work from a to a provided (x > 1) do { Max = 0; }
```

The list of possible warning to silence is given below:

- nondeterminism
- unboundcomponentport
- unboundconnectorport
- missingup
- atomprioritycycle

As we don’t provide any support for serializing our `my_custom_type` data type, we need to turn off the generation of serialization code in atoms:

```
$ bipc.sh -I . -p HelloPackage -d "HelloCompound()"  
  --gencpp-output output 
  --gencpp-cc-I $PWD/ext-cpp 
  --gencpp-no-serial 
$ mkdir output/build 
$ cd output/build 
$ cmake ..
[...]
$ make 
[...]
```

When executing, we get the following trace:

```
[BIP ENGINE]: initialize components...
Data for: 1 = 2,8
Data for: 2 = 4,16
```

As we don’t provide any support for serializing our `my_custom_type` data type, we need to turn off the generation of serialization code in atoms:
4.3.2 Hints on using package

A package named "a.b.c.D" must be stored in a directory hierarchy "a/b/c/D.bip". Anything else will not work. If you want to use packages located outside of your current working directory, you must use the "-I" parameter to add the directories that contain them. For example:

• you are developing in "/somewhere/myApp" a BIP package named "Foo"

• you want to use the package "my.other.package.Bar" located in "/a/bip/repository" directory

Here's the tree snapshot and the corresponding compiler command to use:

```
|-- a
|   |-- bip
|   |   |-- repository
|   |   |-- my
|   |   |-- other
|   |   |-- package
|   |       |-- Bar.bip
|   `-- somewhere
    `-- myApp
```

somewhere/myApp $ bipc.sh -p Foo -I /a/bip/repository

4.4 Using middle-ends (aka. filters)

Filters are responsible for model to model transformations. A filter has the same input and output type: a BIP model (type or instance model). Common use cases for filters:

• flattening: remove hierarchy by flattening compound and connectors.

• dead code optimization: modify Petri net by removing unused parts.

• annotation: attach extra information on model element used by other filters or back-ends.

A filter can be used alone or a filter chain can be build. The chain is specified using a simple syntax:

```
filter1_name foo=bar foo2=bar2 ! filter2_name bla=bar
```

This will chain filter1_name and filter2_name. Each filter will be configured using its corresponding list of key=value pairs.

The chain specification can be given directly on from the command line using -f (or --filter):

```
bipc.sh -f "filter1_name foo=bar foo2=bar2 ! filter2_name bla=bar"
```

4.4.1 Using middle-ends (aka. filters) 35

When executing, we can see that the transition for the positive transition is fired:

```
[BIP ENGINE]: initialize components...
[BIP ENGINE]: state #0: 1 internal port: [0] ROOT.c1._iport_decl__p
[BIP ENGINE]: -> choose [0] ROOT.c1._iport_decl__p
[BIP ENGINE]: state #1: 1 internal port: [0] ROOT.c1._iport_decl__negative
[BIP ENGINE]: -> choose [0] ROOT.c1._iport_decl__negative
```

Someone says: Positive data with data=999

```
[BIP ENGINE]: state #2: deadlock!
```

7.6.6 Using custom type

We will now use custom type in a simple rendez-vous example involving 3 atoms. The expected behavior is very simple:

• each atom calls the init_data() function to initialize its internal data. All atoms get different values.

• they all synchronize and the connector takes the values from the 3rd atom and writes it in the other 2 atoms.

The atoms display their data before and after the synchronization.

For using a custom type, we need:

• to declare the type in the BIP source

• to define the type in the C++ extern code

In this example, we don't provide serialization support (this will be demonstrated in the next example).
Important: Do not forget to enclose the chain specification between " or ', as the shell will most certainly interpret the ! character, leading to unwanted behavior.

The chain specification can also be read from a file using --filter-file. This is useful when the chain is getting complex as handling very long lines can be tedious work. You simply need to write the chain in a text file. To enhance readability, you can use a filter by line convention, as the line feed is ignored:

```
filter1_name foo=bar foo2=bar2
filter2_name bla=bar
filter3_name some_complex_arg=something_very_long
```

And simply give this file to the compiler:

```
blpc.sh --filter-file filters.txt ...
```

### 4.5 Using back-ends (code generators)

#### 4.5.1 General principles

A back-end (aka. code generator) defines a set of specific parameters. Usually, using one of them will enable the corresponding back-end. For example, for the C++ back-end, you can see the following command line arguments (see More about C++ code generator):

```
--gencpp-cc-I
--gencpp-cc-extra=src
--gencpp-failow-used-packages
--gencpp-ld=L
--gencpp-ld-extra-obj
--gencpp-ld-l
--gencpp-no-serial
--gencpp-output-dir
--gencpp-optim
--gencpp-set-optim-param
--gencpp-disable-optim
--gencpp-enable-optim
--gencpp-enable-bip-debug
```

Calling the compiler using any on these parameter will enable the C++ back-end.

You can use more than one back-end at once without any problem as back-end are meant to be independent. For example, for generating both a C++ and Aseba source code in a single compiler run, you could use the following command:

```
blpc.sh --cpp=ext-cpp --filter-file filters.txt ...
```

### 7.6.5 Hello World with external code called from const context

When calling function from const context (eg. connector's up, all guards), one must take extra care when interfacing the external code using data. Again, we extend our `HelloPackage` by adding a guard calling an external function called `my_guard()` that accesses the component's data.

The new BIP:

```cpp
@cpp(src="ext-cpp/HelloPackage.cpp",include="HelloPackage.hpp")
package HelloPackage
    extern function bool my_guard(int)
    extern function my_modify(int)
    extern function my_print(string, int)

    port type HelloPort_t()
    atom type HelloAtom()
    data int somedata
    port HelloPort_t p{, positive{, negative{, initial to START do { somedata = 0; } on p from START to S do { my_modify(somedata); } on negative from S to END provided { my_guard(somedata) } do { my_print("Positive data", somedata); } on positive from S to END provided { my_guard(somedata) } do { my_print("Negative data", somedata); }} end

    component HelloAtom c1()
```

Note that the new `HelloPackage.hpp` includes the declaration of `const_my_guard()` and not `my_guard()`. This is because our BIP calls `my_guard()` from a const context:

```
void my_print(const char *message, int &adata);
void my_modify(int &adata);
bool const_my_guard(int &adata);
```

The compilation is still the same:

```
blpc.sh --cpp=ext-cpp --filter-file filters.txt ...
```
The BIP back-end can be used to generate back BIP source code. It is very simple and uses two parameters:

- `--genbip-output-dir`: to specify the directory where the generated code will be created.
- `--genbip-follow-used-packages`: to enable the hierarchical generation. By default, only the package being compiled is generated back to BIP source code. When this parameter is present, the package’s dependencies are also generated.

If no transformation is executed in the middle-end, then this back-end should produce a source code equivalent to the source code compiled (some code reformating and reordering is very likely to happen):

```
$ bipc.sh -p SamplePackage -I /home/a_user/my_bip_lib/ --genbip-output-dir bip-output
```

Important: This back-end only supports type model compilation. It won't use the instance model that the compiler may produce (if a `-d` parameter is used).

### 4.5.3 C++ back-end

Simple case, for compiling the package `SomePackage` and creating an executable by taking an instance of the `RootDefinition` component use the following command:

```
$ bipc --gencpp-output build -p SomePackage -d 'RootDefinition()'
```

This command will generate several files, mainly C++ source code, but not only. This code can’t be compiled as is, it needs some glue code from a standard engine. See More about C++ code generator for more details on this back-end.

### 4.5. Using back-ends (code generators)

The new `HelloPackage.hpp`:

```cpp
void my_print(const char *message, int &adata);
void my_modify(int &adata);
```

And the corresponding `HelloPackage.cpp`:

```cpp
#include <iostream>

void my_print(const char *message, int &adata){
    std::cout << "Someone says: " << message << " with data= " << adata << std::endl;
}

void my_modify(int &adata){
    adata = 999;
}
```

The compilation is still the same:

```
$ bipc.sh -I . -p HelloPackage -d "HelloCompound()" --gencpp-output output --gencpp-cc-I $PWD/ext-cpp
```

```
$ mkdir output/build
$ cd output/build
$ cmake ..
```

```
[...]
$ make
[...]
```

When running the example, we can see the integer is correctly modified:

```
$ ./system
[BIP ENGINE]: initialize components...
[BIP ENGINE]: state #0: 1 internal port:
[BIP ENGINE]: [0] ROOT.c1._iport_decl__p
[BIP ENGINE]: -> choose [0] ROOT.c1._iport_decl__p
[BIP ENGINE]: state #1: 1 internal port:
[BIP ENGINE]: [0] ROOT.c1._iport_decl__p
[BIP ENGINE]: -> choose [0] ROOT.c1._iport_decl__p
Someone says: Hello World with data=999
[BIP ENGINE]: state #2: deadlock!
```

84 Chapter 7. Tutorial
data int somedata
port HelloPort_t p()
place START,END
initial to START do { somedata = 0; }
on p from START to END do (my_print("Hello World", somedata));
end

compound type HelloCompound()
    component HelloAtom cl()
end
end

The `my_print()` is changed to accept an extra `int` parameter. Note that this parameter is a C++ reference: the function has access to the real data, not a copy.

HelloPackage.hpp:

```cpp
#include <iostream>

void my_print(const char *message, int &adata);
```

HelloPackage.cpp:

```cpp
#include <iostream>

void my_print(const char *message, int &adata){
    std::cout << "Someone says: " << message << " with data= " << adata << std::endl;
}
```

The compilation is still the same:

```bash
$ bipc.sh -I . -p HelloPackage -d "HelloCompound()"
--gencpp-output output 
--gencpp-cc-I $PWD/ext-cpp
$ mkdir output/build
$ cd output/build
$ cmake ..
[...]
$ make
[...]
```

When running the executable, we can see that the value for the data is correctly display:

```
[BIP ENGINE]: initialize components...
[BIP ENGINE]: state #0: 1 internal port:
[BIP ENGINE]:  [0] ROOT.c1._iport_decl_p
[BIP ENGINE]:  -> choose [0] ROOT.c1._iport_decl_p
Someone says: Hello World with data=0
[BIP ENGINE]: state #1: deadlock!
```

### 7.6.4 Hello World with data modified by external code

The previous example simply shows how to read data received from BIP inside external code. The external code can also modify this code (if called from a context that allows the modification of the data). We add a new `my_modify()` function in our external code that only modifies its integer parameter.

The new BIP code:

```cpp
@cpp(src="ext-cpp/HelloPackage.cpp",include="HelloPackage.hpp")
package HelloPackage
    extern function my_modify(int)
```
5. Presentation & prerequisites

5.1 Presentation

The C++ back-end produces a set of C++ source files along with a set of CMake scripts used to compile the generated C++ files and link them with an engine.

5.1.2 Prerequisites

In order to use the code generated by the C++ back-end, you need to install the following dependencies:

- CMake, at least version 2.8.2. It may work with earlier versions, but it has not been tested.
- GNU Make.
- A C++ compiler that supports the STL. In addition, a support for C++0x is required when compiling with the optimized engine, and C++11 for the multithread engine. We are currently working with the GNU compiler g++ version 4.8.2, and for ABI compatibility issues we recommend to use g++ version 4.8 or higher.

Tip:

On GNU/Debian or derivatives, use:

$ apt-get install cmake make g++

5.2 Usage

To generate C++ code, extra parameters must be used to drive C++ code generation.

Important:

If you are not using the standard compiler distribution, then you need to take care of the correct loading of the C++ back-end: its jar file must be in the classpath and the java property `bip.compiler.backends` must contain the string `ujf.verimag.bip.backend.cpp.CppBackend`.

The current C++ code generation requires the presence of an instance model, thus you must provide a root declaration (see `-d` in the above section). To enable the C++ back-end, you simply need to give an output directory:

- `--gencpp-output-dir` followed by the directory that will contain all files generated by the C++ back-end.

Example:

$ bipc.sh -p SamplePackage -I /home/a_user/my_bip_lib/ -d "MyType()" --gencpp-output-dir /home/a_user/output/

The `--gencpp-cc-I` is used to include the directory containing our `.hpp` file to the C++ compiler include paths.

And finally, run the produced system executable:

$ ./system

[BIP ENGINE]: initialize components...
[BIP ENGINE]: state #0: 1 internal port:
[BIP ENGINE]: [0] ROOT.c1._iport_decl__p
[BIP ENGINE]: -> choose [0] ROOT.c1._iport_decl__p
Someone says: Hello World
[BIP ENGINE]: state #1: deadlock!

7.6.3 Hello World with data and external code

In this example, we modify again our Hello World, this time to pass some data to the external code.

The new BIP code is now:

```cpp
@cpp(src="ext-cpp/HelloPackage.cpp",include="HelloPackage.hpp")
package HelloPackage

extern function my_print(string, int)
port type HelloPort_t()
atom type HelloAtom()
```

8.2 Usage

The code generator can be used to drive the C++ code generation:

- `--gencpp-output-dir` followed by the directory that will contain all files generated by the C++ back-end.

Example:

$ bipc.sh -p SamplePackage -I /home/a_user/my_bip_lib/ -d "MyType()" --gencpp-output-dir /home/a_user/output/

The `--gencpp-cc-I` is used to include the directory containing our `.hpp` file to the C++ compiler include paths.

And finally, run the produced system executable:

$ ./system

[BIP ENGINE]: initialize components...
[BIP ENGINE]: state #0: 1 internal port:
[BIP ENGINE]: [0] ROOT.c1._iport_decl__p
[BIP ENGINE]: -> choose [0] ROOT.c1._iport_decl__p
Someone says: Hello World
[BIP ENGINE]: state #1: deadlock!
The directory `/home/a_user/output` should contain several files & directories:
```
÷-- CMakeLists.txt
÷-- Deploy
| ÷-- Deploy.cpp
| | ++ Deploy.hpp
| | ++ DeployTypes.hpp
÷-- SamplePackage
| ++ CMakeLists.txt
| | ++ include
| | | ++ CT_MyType.hpp
| | | ++ AtomEPort_Port__t.hpp
| | | ++ AtomIPort_Port__t.hpp
| | -- src
| | -- SamplePackage
| | | -- CT_MyType.cpp
| | | -- AtomEPort_Port__t.cpp
| | | -- AtomIPort_Port__t.cpp
```
You don’t need to dig into these directories, but it’s always better to understand how the compiler organizes the generated files:
- a master `CMakeLists.txt` that will be used to compile and link everything (generated code and engine code) together. Its use will be demonstrated later.
- a directory `SamplePackage` containing:
  - a `CMakeLists.txt` with directives to compile the package
  - an `include` directory with all `header` files (ie. `.hpp` files).
  - a `src` directory with all implementation files (ie. `.cpp` files).
- a directory `Deploy` with a 3 files with the directives for the concrete deployment of the running system.

By default, the compiler won’t resolve dependencies and will fail in case of inter-package reference. You need to provide `--gencpp-follow-used-packages` to resolve and compile dependencies.

### 5.3 Interface BIP/C++

#### 5.3.1 Presentation

It is very common to interface BIP code with external C++ code (eg. legacy code, specific code, ...). The current back-end provides you with several ways to interface your BIP code with external C++ code.

Both ways of interfacing may need to add directory to the C++ compiler include search path. This can be achieved by using this command line argument:
```
• --gencpp-cc-I
```

At the package/type level

You can add one or more source file (ie. `.cpp` file) or object file (ie. `.o` file) attached to a package/a type. These source file will be compiled at the same time as the generated files corresponding to the package/type and the object

When running the example, you can see our `printf()` being executed when the transition is fired:

```
[BIP ENGINE]: initialize components...
[BIP ENGINE]: state #0: 1 internal port:
[BIP ENGINE]: -> choose #0 ROOT.c1._iport_decl__p
[BIP ENGINE]: Hello World
[BIP ENGINE]: state #1: deadlock!
```

### 7.6.2 Hello World with external code

Let’s modify again our example. This time, we will also provide the code needed for printing the message to the console instead of relying directly on a `standard library`.

Change the previous `HelloPackage.bip` by adding an extra annotation on the package definition:
```
@cpp(src="ext-cpp/HelloPackage.cpp",include="HelloPackage.hpp")
```

Along with the BIP file, you need to create the external code that will provide the `my_print(...)` function:

```
• the interface (ie. `HelloPackage.hpp`) that you need to put in a directory that will be included in the C++ compiler search path.
```

When the example is executed, you can see our `printf()` being executed when the transition is fired:

```
[BIP ENGINE]: initialize components...
[BIP ENGINE]: state #0: 1 internal port:
[BIP ENGINE]: -> choose #0 ROOT.c1._iport_decl__p
[BIP ENGINE]: Hello World
[BIP ENGINE]: state #1: deadlock!
```

#### 7.6.3 Hello World with external code

Let’s modify again our example. This time, we will also provide the code needed for printing the message to the console instead of relying directly on a `standard library`.

Change the previous `HelloPackage.bip` by adding an extra annotation on the package definition:
```
@cpp(src="ext-cpp/HelloPackage.cpp",include="HelloPackage.hpp")
```

Along with the BIP file, you need to create the external code that will provide the `my_print(...)` function:

```
• the interface (ie. `HelloPackage.hpp`) that you need to put in a directory that will be included in the C++ compiler search path.
```

When the example is executed, you can see our `printf()` being executed when the transition is fired:

```
[BIP ENGINE]: initialize components...
[BIP ENGINE]: state #0: 1 internal port:
[BIP ENGINE]: -> choose #0 ROOT.c1._iport_decl__p
[BIP ENGINE]: Hello World
[BIP ENGINE]: state #1: deadlock!
```
For context where the callee can change the data (ie. provided() is mapped to

\[ \text{const context:} \quad \text{ie. provided()} \]

\[ \text{provided()} \]

\[ \text{const context:} \quad \text{ie. provided()} \]

\[ \text{provided()} \]

\[ \text{const context:} \quad \text{ie. provided()} \]

\[ \text{provided()} \]

\[ \text{const context:} \quad \text{ie. provided()} \]

\[ \text{provided()} \]

\[ \text{const context:} \quad \text{ie. provided()} \]

\[ \text{provided()} \]

\[ \text{const context:} \quad \text{ie. provided()} \]

\[ \text{provided()} \]

\[ \text{const context:} \quad \text{ie. provided()} \]

\[ \text{provided()} \]

\[ \text{const context:} \quad \text{ie. provided()} \]

\[ \text{provided()} \]

\[ \text{const context:} \quad \text{ie. provided()} \]

\[ \text{provided()} \]

\[ \text{const context:} \quad \text{ie. provided()} \]

\[ \text{provided()} \]

\[ \text{const context:} \quad \text{ie. provided()} \]

\[ \text{provided()} \]

\[ \text{const context:} \quad \text{ie. provided()} \]

\[ \text{provided()} \]

\[ \text{const context:} \quad \text{ie. provided()} \]

\[ \text{provided()} \]

\[ \text{const context:} \quad \text{ie. provided()} \]

\[ \text{provided()} \]

\[ \text{const context:} \quad \text{ie. provided()} \]

\[ \text{provided()} \]

\[ \text{const context:} \quad \text{ie. provided()} \]

\[ \text{provided()} \]

\[ \text{const context:} \quad \text{ie. provided()} \]

\[ \text{provided()} \]

\[ \text{const context:} \quad \text{ie. provided()} \]

\[ \text{provided()} \]

\[ \text{const context:} \quad \text{ie. provided()} \]

\[ \text{provided()} \]

\[ \text{const context:} \quad \text{ie. provided()} \]

\[ \text{provided()} \]

\[ \text{const context:} \quad \text{ie. provided()} \]

\[ \text{provided()} \]

\[ \text{const context:} \quad \text{ie. provided()} \]

\[ \text{provided()} \]

\[ \text{const context:} \quad \text{ie. provided()} \]

\[ \text{provided()} \]

\[ \text{const context:} \quad \text{ie. provided()} \]

\[ \text{provided()} \]

\[ \text{const context:} \quad \text{ie. provided()} \]

\[ \text{provided()} \]

\[ \text{const context:} \quad \text{ie. provided()} \]

\[ \text{provided()} \]

\[ \text{const context:} \quad \text{ie. provided()} \]

\[ \text{provided()} \]

\[ \text{const context:} \quad \text{ie. provided()} \]

\[ \text{provided()} \]

\[ \text{const context:} \quad \text{ie. provided()} \]

\[ \text{provided()} \]

\[ \text{const context:} \quad \text{ie. provided()} \]

\[ \text{provided()} \]

\[ \text{const context:} \quad \text{ie. provided()} \]

\[ \text{provided()} \]

\[ \text{const context:} \quad \text{ie. provided()} \]

\[ \text{provided()} \]

\[ \text{const context:} \quad \text{ie. provided()} \]

\[ \text{provided()} \]

\[ \text{const context:} \quad \text{ie. provided()} \]

\[ \text{provided()} \]

\[ \text{const context:} \quad \text{ie. provided()} \]

\[ \text{provided()} \]

\[ \text{const context:} \quad \text{ie. provided()} \]

\[ \text{provided()} \]

\[ \text{const context:} \quad \text{ie. provided()} \]

\[ \text{provided()} \]

\[ \text{const context:} \quad \text{ie. provided()} \]

\[ \text{provided()} \]

\[ \text{const context:} \quad \text{ie. provided()} \]

\[ \text{provided()} \]

\[ \text{const context:} \quad \text{ie. provided()} \]

\[ \text{provided()} \]

\[ \text{const context:} \quad \text{ie. provided()} \]

\[ \text{provided()} \]

\[ \text{const context:} \quad \text{ie. provided()} \]

\[ \text{provided()} \]

\[ \text{const context:} \quad \text{ie. provided()} \]

\[ \text{provided()} \]

\[ \text{const context:} \quad \text{ie. provided()} \]

\[ \text{provided()} \]

\[ \text{const context:} \quad \text{ie. provided()} \]

\[ \text{provided()} \]

\[ \text{const context:} \quad \text{ie. provided()} \]

\[ \text{provided()} \]

\[ \text{const context:} \quad \text{ie. provided()} \]

\[ \text{provided()} \]

\[ \text{const context:} \quad \text{ie. provided()} \]

\[ \text{provided()} \]

\[ \text{const context:} \quad \text{ie. provided()} \]

\[ \text{provided()} \]

\[ \text{const context:} \quad \text{ie. provided()} \]

\[ \text{provided()} \]

\[ \text{const context:} \quad \text{ie. provided()} \]

\[ \text{provided()} \]

\[ \text{const context:} \quad \text{ie. provided()} \]

\[ \text{provided()} \]

\[ \text{const context:} \quad \text{ie. provided()} \]

\[ \text{provided()} \]

\[ \text{const context:} \quad \text{ie. provided()} \]

\[ \text{provided()} \]

\[ \text{const context:} \quad \text{ie. provided()} \]

\[ \text{provided()} \]

\[ \text{const context:} \quad \text{ie. provided()} \]

\[ \text{provided()} \]

\[ \text{const context:} \quad \text{ie. provided()} \]

\[ \text{provided()} \]

\[ \text{const context:} \quad \text{ie. provided()} \]

\[ \text{provided()} \]

\[ \text{const context:} \quad \text{ie. provided()} \]

\[ \text{provided()} \]

\[ \text{const context:} \quad \text{ie. provided()} \]

\[ \text{provided()} \]

\[ \text{const context:} \quad \text{ie. provided()} \]

\[ \text{provided()} \]

\[ \text{const context:} \quad \text{ie. provided()} \]

\[ \text{provided()} \]

\[ \text{const context:} \quad \text{ie. provided()} \]

\[ \text{provided()} \]

\[ \text{const context:} \quad \text{ie. provided()} \]

\[ \text{provided()} \]

\[ \text{const context:} \quad \text{ie. provided()} \]

\[ \text{provided()} \]

\[ \text{const context:} \quad \text{ie. provided()} \]

\[ \text{provided()} \]

\[ \text{const context:} \quad \text{ie. provided()} \]

\[ \text{provided()} \]

\[ \text{const context:} \quad \text{ie. provided()} \]

\[ \text{provided()} \]

\[ \text{const context:} \quad \text{ie. provided()} \]

\[ \text{provided()} \]

\[ \text{const context:} \quad \text{ie. provided()} \]

\[ \text{provided()} \]

\[ \text{const context:} \quad \text{ie. provided()} \]

\[ \text{provided()} \]

\[ \text{const context:} \quad \text{ie. provided()} \]

\[ \text{provided()} \]

\[ \text{const context:} \quad \text{ie. provided()} \]

\[ \text{provided()} \]

\[ \text{const context:} \quad \text{ie. provided()} \]

\[ \text{provided()} \]

\[ \text{const context:} \quad \text{ie. provided()} \]

\[ \text{provided()} \]

\[ \text{const context:} \quad \text{ie. provided()} \]

\[ \text{provided()} \]

\[ \text{const context:} \quad \text{ie. provided()} \]

\[ \text{provided()} \]

\[ \text{const context:} \quad \text{ie. provided()} \]

\[ \text{provided()} \]

\[ \text{const context:} \quad \text{ie. provided()} \]

\[ \text{provided()} \]

\[ \text{const context:} \quad \text{ie. provided()} \]

\[ \text{provided()} \]

\[ \text{const context:} \quad \text{ie. provided()} \]

\[ \text{provided()} \]
does not take const argument, it will still work, but system data may be altered by error. The const-ness is not a guaranty, it's only a good guide that avoids making mistakes.

- \( x = f(a, b, c) \) mapped to \( \text{internal\_data\_x = const\_f(internal\_data\_a, internal\_data\_b, internal\_data\_c)\text{, with expected prototype: T1 f(const T2 &a, const T3 &b, const T4 &c)}\)\). Beware that the return type is not a reference nor a pointer. This in order to avoid useless copy.

Hint: C++ code generator uses different function names instead of relying on C++ dispatching mechanism because it doing so would imply that the compiler is able to type function parameters, which is currently not the case.

Important: When using custom types, you may run into problems when using the reference engine as it tries to display a serialized version of the data during execution. This serialization relies on the C++ stream mechanism. If your data type does not support stream operation, the generated code won’t compile. You can disable serialization when running the compiler with --gencpp-no-serial (no data will be displayed in execution traces).

The Using the C++ back-end has examples of BIP/C++ interfacing.

Handling component parameter

If you need to use a component parameter in an external function call, the parameter in the function prototype must not be a reference. Treat component parameters as direct value or expression:

```c
atom type AT(int x)
...
  on p from S to T do {f(x);}
...
end
```

The function must look like:

```c
void f(int x);
```

If you try to use a reference, the C++ compiler will fail.

Pass by reference/copy

When an external function takes a data variable (ie. atom data, component exported data, connector data) as parameter, do not forget to use a reference in the function prototype. Even if omitted, the code will still compile flawlessly, but the function will work on a copy of the data variable, not the variable itself. Any modification will be lost and strange behavior can arise because of the unwanted use of the copy constructor.

If the function is given a data from a component type parameter or a direct value, then the corresponding function parameter must not be a reference.

For example:

```c
atom type AT()
data int x
...
  on p from S to T do {f(x);}
...
end
```

\( f \) should have the following prototype:

```c
void f(int x);
```

When compiling and executing an instance of Model, we obtain an execution in which only component B is executing. This comes from the fact that the transition from place FREE to place WAIT in B is internal, that is, it is the state of B before the its execution is invisible. As a result, interactions of A_utilize_R can never executes since the visible value of B.free is always 0.

...[BIP ENGINE]: initialize components...
[BIP ENGINE]: state #0: 1 interaction:
[BIP ENGINE]: [0] ROOT.B_utilize_R: ROOT.B.utilize( ROOT.R.utilize())
[BIP ENGINE]: state #1: 1 interaction:
[BIP ENGINE]: [0] ROOT.B_utilize_R: ROOT.B.utilize( ROOT.R.utilize())
[BIP ENGINE]: -> choose [0] ROOT.R_utilize_R: ROOT.R.utilize( ROOT.B.utilize())
[BIP ENGINE]: state #2: 1 interaction:
[BIP ENGINE]: [0] ROOT.B_utilize_R: ROOT.B.utilize( ROOT.R.utilize())
[BIP ENGINE]: -> choose [0] ROOT.R_utilize_R: ROOT.R.utilize( ROOT.B.utilize())
[BIP ENGINE]: state #3: 1 interaction:
[BIP ENGINE]: [0] ROOT.B_utilize_R: ROOT.B.utilize( ROOT.R.utilize())
[BIP ENGINE]: -> choose [0] ROOT.R_utilize_R: ROOT.R.utilize( ROOT.B.utilize())
...
To enable optimisation of the set of interactions defined at the top of the file:

```cpp
# BIPHP
BIPHP

[Preprocessor Options] [Type] [Base]

--gencpp-cc-I
--gencpp-cc-extra-src
--gencpp-ld-L
--gencpp-ld-l
--gencpp-ld-extra-obj
--gencpp-follow-used-packages
--gencpp-no-serial
--gencpp-disable-optim
--gencpp-enable-optim
--gencpp-optim
--gencpp-set-optim-param
--gencpp-enable-bip-debug
```

5.5 Optimisation

The C++ back-end can apply some optimisation techniques. You can enable them either one by one, or by using predefined groups.

To enable all optimizations up to level 2:

```
$ bipc.sh ... --gencpp-optim 2
```

To enable the use of a pool of interaction object of size 200:

```
--gencpp-optim-pool-size 200
```

### Dynamic priorities and invisible states

In the following example, the components `A` and `B` represent potential users of a resource which is represented by the component `R`. When a user `A` or `B` reaches the place `FREE`, it sets its variable `free` to 1 which is exported to inform that it is not using the resource `R`. The variable `free` or a user is set to 0 when it leaves the place `FREE` to inform that it reaches the place `WAIT` from which it may use the resource. To prevent from concurrent use of the resource, a scheduler has been implemented using priorities, as explained as follows. It gives more priority to `B` provided `B` is in place `FREE`, that is, its variable `free` equals to 0. Notice that use of `*` in the priority rule: it gives less priority to interactions of defined in `A_utilize_R` than any interaction defined in any connector except `A_utilize_R`.

```bip
package priorities_invisible
port type Port()
atom type Resource()
export port Port utilize()
place WAIT
initial to WAIT
on utilize from WAIT to WAIT
end
atom type UserOfResource()
export data int free
export port Port utilize()
```
Currently, the following optimizations are available:

- `rdvconnector` (level: 1): generates specific code for rendez-vous connectors.
- `poolci` (level: 2): dynamically created interaction object can be reused. When released, an interaction is placed in a pool. When a lot of interactions are involved, it lightens the burden on the memory allocator. The cost is that some memory is never released.
- `poolciv` (level: 2): same as `poolci` but for interaction value objects.
- `ports-reset` (level: 2): allows to reduce recomputation of interactions and internal ports after components execution, based on static analysis of the code executed by transitions of atomic components. This optimization is only exploited by the optimized engine (i.e. no gain when using the reference engine).
- `no-side-effect` (level: 3): improves other optimizations (currently concerns only optimization `ports-reset`) by assuming that assignments of a variable v of an external type only modify v (e.g. no side effect on any other variable due to aliasing), and that calls to external functions can only modify the variables provided as parameters.

Both `poolci` and `poolciv` accepts an optional parameter `size` to set the size of the pool. Beware that a pool of fixed size is created for every connector instance.

### 5.6 Debugging

BIP tools do not include a full featured debugger. Instead, we provide a mapping between the generated C++ code (on which any C++ debugger can be used) and the BIP source code. To enable this mechanism, you need to compile the code using `--gencpp-enable-bip-debug`.

The direct benefits are:

- use of breakpoints in BIP source code
- step by step execution in BIP source code

The direct drawbacks are:

- it is not possible to print data using BIP variable names, you need to dig into the generated code, which is less easy since it is the BIP code that gets displayed.
- inconcoherances/unexpected debugger behavior can appear, as the mapping is not necessarily bijective (e.g. a BIP guard could be duplicated in two locations in the generated code)

Important: You need to compile the C++ with debugging support. Use the Debug profile included in the cmake script:

```
$ cmake -DCMAKE_BUILD_TYPE=Debug
```

### 5.7 Annotations

#### 5.7.1 `@cpp(src="<file-list>")`

- `scope`: package definition, any type definition

#### Using priorities to enforce an order of execution

We can modify the previous example to enforce the execution of the interaction 'D.p,E.p,F.p' of `brdDEF` before the execution of the interaction 'A.p,B.p' of `brdABC`. For this, we add the following priority rule in Model:

```
```

This ensures that the model has a single execution sequence which is the following:

---

BIP2 Documentation, Release 2015.04 (RC7)

$ bipc.sh ... --gencpp-enable-optim poolci \n--gencpp-set-optir-paixan poolci:size=2

Currently, the following optimizations are available:

- `rdvconnector` (level: 1): generates specific code for rendez-vous connectors.
- `poolci` (level: 2): dynamically created interaction object can be reused. When released, an interaction is placed in a pool. When a lot of interactions are involved, it lightens the burden on the memory allocator. The cost is that some memory is never released.
- `poolciv` (level: 2): same as `poolci` but for interaction value objects.
- `ports-reset` (level: 2): allows to reduce recomputation of interactions and internal ports after components execution, based on static analysis of the code executed by transitions of atomic components. This optimization is only exploited by the optimized engine (i.e. no gain when using the reference engine).
- `no-side-effect` (level: 3): improves other optimizations (currently concerns only optimization `ports-reset`) by assuming that assignments of a variable v of an external type only modify v (e.g. no side effect on any other variable due to aliasing), and that calls to external functions can only modify the variables provided as parameters.

Both `poolci` and `poolciv` accepts an optional parameter `size` to set the size of the pool. Beware that a pool of fixed size is created for every connector instance.

### 5.6 Debugging

BIP tools do not include a full featured debugger. Instead, we provide a mapping between the generated C++ code (on which any C++ debugger can be used) and the BIP source code. To enable this mechanism, you need to compile the code using `--gencpp-enable-bip-debug`.

The direct benefits are:

- use of breakpoints in BIP source code
- step by step execution in BIP source code

The direct drawbacks are:

- it is not possible to print data using BIP variable names, you need to dig into the generated code, which is less easy since it is the BIP code that gets displayed.
- inconcoherances/unexpected debugger behavior can appear, as the mapping is not necessarily bijective (e.g. a BIP guard could be duplicated in two locations in the generated code)

Important: You need to compile the C++ with debugging support. Use the Debug profile included in the cmake script:

```
$ cmake -DCMAKE_BUILD_TYPE=Debug
```

### 5.7 Annotations

#### 5.7.1 `@cpp(src="<file-list>")`

- `scope`: package definition, any type definition

#### Using priorities to enforce an order of execution

We can modify the previous example to enforce the execution of the interaction 'D.p,E.p,F.p' of `brdDEF` before the execution of the interaction 'A.p,B.p' of `brdABC`. For this, we add the following priority rule in Model:

```
```

This ensures that the model has a single execution sequence which is the following:
6.8 What you should never do

In this section, we give examples of things you should never do. All these examples will compile and run, and sometimes have the behavior you expected. But they all break at least one of the strong assumptions on which BIP is based.

7.5.2 Priorities in compounds

Similarly to the use of priorities in atoms, when several interactions are enabled at a given state of a compound, priorities can be used to prevent some of them from executing.

example:

```bip
connector type Broadcast(Port p, Port q, Port r)
define p' q r
```
5.8.1 Non-deterministic external code

The most simple example of a non-deterministic code is the use of standard library’s random() function.

For example, consider the following package:

```cpp
@cpp(include="stdio.h,stdlib.h")
package bad
  port type Port_t()
  atom type BadAtom()
  data int d
  port Port_t p()
  place I,S1,S2
  initial to I do { d = 0; }
  on p from I to S1 do { d = random()%5; }
  on p from S1 to S1 provided (d > 0) do { d = d - 1; }
  on p from S1 to S2 provided (d <= 0)
end
compound type Top()
  component BadAtom c()
end
end
```

The following assumption: “From a given system state (here, atom c in state I and d equals 0), triggering a transition t always transforms the system state in the same state (here, atom c in state S1 with d equals some value)” is broken. Even if there is only one single transition possible in the petrinet from state”I” to S1, the system state remains unknown as the value for d is not always the same.

Even if this may be the expected behavior, this is a problem when verification tools are used. For example, the exploration heavily relies on the assumption being broken and thus, will produce incorrect results for this example.

5.8.2 Side-effects in guards or up{}

As explained earlier, all guards and connector up{} must not have side effects on the system. This is very important, as the engine may execute several times these methods or it may cache their results: you can’t predict how these will be executed.

The BIP compiler prevents the user from writing wrong statements, but as always when using external code, it is still possible to make mistake.

The following example illustrates both cases:

- the guard() method, that should not modify its data parameter will in fact modify them by calling wrong_guard_ip()
- the up{} will also call a function wrong_up() that will modify data bound to the connector’s ports.

Such an example demonstrates both a wrong execution and incorrect verification results:

```
package priorities_in_atom
  port type Port()
  atom type MyAtom()
  data int i
  port Port p(), q()
  place LOOP
  initial to LOOP do { i=0; }
  on p from LOOP to LOOP do { i=i+1; }
  on q from LOOP to LOOP do { i=i+1; }
  priority myPrioEven q < p provided ((i%2) == 0)
  priority myPrioOdd p < q provided ((i%2) == 1)
end
compound type Model()
  component MyAtom a()
end
```

Notice that the compilation of the previous BIP2 code leads to the following warning due to the potential cycle in priorities introduced by the rules myPrioEven and myPrioOdd:

```
[WARNING] In /home/to/example/priorities_in_atom.bip:
Cycle found in priorities in Atom type :
18:
  19:    priority myPrioEven q < p provided ((i%2) == 0)  -------------
20:    priority myPrioOdd p < q provided ((i%2) == 1)
21: end
```

Priorities may also be defined dynamically using guards involving variables. In this case, cycles are checked at runtime. An example of dynamic priority can be found in the following section.

Using priorities to enforce an order of execution

We can also modify the previous example to execute both transitions labelled by ports p and q, but with imposing the order of execution by using priorities. Assume we want to enforce that p and q are alternately executed, starting by p. For this, we first add an integer variable i representing the state number of the atom, that is, it is initialized at 0 and incremented every transition execution. We also give more priority to p for even state numbers, and more priority for q for odd state numbers.

```
package priorities_in_atom
  port type Port()
  atom type MyAtom()
  data int i
  port Port p(), q()
  place LOOP
  initial to LOOP do { i=0; }
  on p from LOOP to LOOP do { i=i+1; }
  on q from LOOP to LOOP do { i=i+1; }
  priority myPrioEven q < p provided ((i%2) == 0)
  priority myPrioOdd p < q provided ((i%2) == 1)
end
compound type Model()
  component MyAtom a()
end
```

Notice that the compilation of the previous BIP2 code leads to the following warning due to the potential cycle in priorities introduced by the rules myPrioEven and myPrioOdd:

```
[WARNING] In /home/to/example/priorities_in_atom.bip:
Cycle found in priorities in Atom type :
18:
  19:    priority myPrioEven q < p provided ((i%2) == 0)  -------------
20:    priority myPrioOdd p < q provided ((i%2) == 1)
21: end
```
In this case, only the transition corresponding to the internal port `p` can be executed. Notice that in this case the model defines a single execution sequence, which is the following:

\[ \text{initial to LOOP do } \{ i = 0; \} \]

\[ \text{on } p \text{ from LOOP to LOOP do } \{ i = i + 1; \} \]

\[ \text{on } q \text{ from LOOP to LOOP do } \{ i = i + 1; \} \]

\[ \text{on } r \text{ from NON-REACHABLE to NON-REACHABLE } \]

\[ \text{priority } \text{myPrio1 } q < r \]

\[ \text{priority } \text{myPrio2 } r < p \]

With `sideeffects.hpp` containing:

```cpp
static void const_wrong_up(int &px)
{ px = -1; }

static int const_wrong_guard_ip(int &d)
{ d = -1; return 0; }
```

The associated execution trace illustrates clearly the problem regarding the `wrong_guard_ip()` statement. Even though the transition labeled by `ip2` is never possible, it can go undetected and, as a result, internal data is modified. When the transition is enabled, the system can run into a state where the data has been wrongly modified and cannot change back.

5.8. What you should never do

In this case, only the transition corresponding to the internal port `p` can be executed. Notice that in this case the model defines a single execution sequence, which is the following:

\[ \text{initial to LOOP do } \{ i = 0; \} \]

\[ \text{on } p \text{ from LOOP to LOOP do } \{ i = i + 1; \} \]

\[ \text{on } q \text{ from LOOP to LOOP do } \{ i = i + 1; \} \]

\[ \text{on } r \text{ from NON-REACHABLE to NON-REACHABLE } \]

\[ \text{priority } \text{myPrio1 } q < p \]

\[ \text{priority } \text{myPrio2 } r < p \]

Notice that a set of rules may define a cyclic relation. Adding the rule `priority myPrio3 p < q` to `MyAtom` in the previous example leads to following error raised by the BIP2 compiler:

```
74 Chapter 7. Tutorial
```

Notice that a set of rules may define a cyclic relation. Adding the rule `priority myPrio3 p < q` to `MyAtom` in the previous example leads to following error raised by the BIP2 compiler:

```
74 Chapter 7. Tutorial
```
The problem with the `wrong_up()` function is more subtle. The value changed is not the atom's data but a port value. This port value is used to compute interactions and evaluate guards of connectors. Modifying it will lead silently to an undefined state (e.g., some interactions may be executed even though their guards should have prevented it).

## 5.9 Troubleshooting

The following is not an exhaustive list of errors with their explanations as most error messages should be self-explained. We give details about more obscure messages that usually deal with low level errors where user friendliness is not the main concern.

### 5.9.1 Assertion `'_iport_decl__p.hasPortValue()'` failed.

If you get an output similar to:

```cpp
system: somepath/HelloPackage/AT_MyAtomType.cpp:141: BipError&
AT_MyAtomType::updatePortValues(): Assertion `'_iport_decl__p.hasPortValue()'` failed.
```

It usually means that an instance of the atom type `MyAtomType` has reached a state where two (or more) transitions labeled by the same port (here `aport`) are possible. You should get a warning at compilation:

```cpp
[WARNING] In path/to/HelloPackage.bip:
Transition from this state triggered by the same port (or internal) already exists:
```

followed by an excerpt of the potentially faulty transition. Chances are that the guards on the transitions labelled by `aport` are not exclusive as they should be.

### 5.9.2 XXXXX.cpp:000: error: `const_SOMETHING` was not declared in this scope

This error is the sign that you have at least of call to the `SOMETHING` function from a const context but the `const_SOMETHING` function implementation could not be found by the C++ compiler.

Check:

- that the external code has the `const_SOMETHING` function, if not, add it.
- if the `const_SOMETHING` function is correctly defined, then check that the search paths given to the C++ are correct (see `--gencpp-cc-I`)

If you think you are not using the function `SOMETHING` from a const context, then, check your BIP code (the `XXXX` in the C++ error message is a hint for a starting point).

### 5.9.3 error: no match for `operator<<`

If you get an error similar to:

```
path/to/AT_AType.cpp: In member function 'virtual std::string AT_AType::toString() const':
path/to/AT_Type.cpp:000: error: no match for 'operator<<' in 'std::operator<<
[with _Traits = std::char_traits<char>]' ...
[C++ garbage]
```

We can modify the following example to prevent from execution of the transition labelled by `q` by simply giving the priority rule `q < p in MyAtom`. We could also use `q < *` which gives less priority to `q` than any other port:

```cpp
package priorities_in_atom
atom type MyAtom()
port Port p(), q()
place LOOP
initial to LOOP
on p from LOOP to LOOP
priority myPrio q < p
end
compound type Model()
component MyAtom a()
end
end
```

The execution of the C++ code obtained from the compilation of an instance of `Model` shows that at each state the two internal ports `p` and `q` can be executed. Thus, the model defines an infinite number of execution sequences. In the standard execution mode of the engine, the choice of the port is made randomly. A typical execution for this example is the following:

```cpp
...
[BIP ENGINE]: initialize components...
[BIP ENGINE]: state #0: 2 internal ports:
[BIP ENGINE]: [0] ROOT.a._iport_decl__p
[BIP ENGINE]: [1] ROOT.a._iport_decl__q
[BIP ENGINE]: -> choose [0] ROOT.a._iport_decl__p
[BIP ENGINE]: state #1: 2 internal ports:
[BIP ENGINE]: [0] ROOT.a._iport_decl__p
[BIP ENGINE]: [1] ROOT.a._iport_decl__q
[BIP ENGINE]: -> choose [1] ROOT.a._iport_decl__q
[BIP ENGINE]: state #2: 2 internal ports:
[BIP ENGINE]: [0] ROOT.a._iport_decl__p
[BIP ENGINE]: [1] ROOT.a._iport_decl__q
[BIP ENGINE]: -> choose [0] ROOT.a._iport_decl__p
[BIP ENGINE]: state #3: 2 internal ports:
[BIP ENGINE]: [0] ROOT.a._iport_decl__p
[BIP ENGINE]: [1] ROOT.a._iport_decl__q
[BIP ENGINE]: -> choose [0] ROOT.a._iport_decl__p
...
```

### Using priorities to inhibit the execution of port `q`

```cpp
package priorities_in_atom
atom type MyAtom()
port Port p(), q()
place LOOP
initial to LOOP
on p from LOOP to LOOP
on q from LOOP to LOOP
priority myPrio q < p
end
compound type Model()
component MyAtom a()
end
end
```
You are probably using data that the compiler can’t deserialize. Two solutions exist for fixing this:

• disable the serialization mechanism by using the --gencpp-no-serial command line argument.

• add serialization support for your type by implementing the operator `<<`.

5.9.4 error: ‘my_XXX’ has a previous declaration
With `my_XXX` being a custom type name or an external function name. This usually means that one of your external header file gets included more than once, hence the duplicated declarations. You should always include guards:

```cpp
#ifndef MY_CUSTOM_FILE_NAME__HPP
#define MY_CUSTOM_FILE_NAME__HPP
[the actual content of the header file]
#endif // MY_CUSTOM_FILE_NAME__HPP
```

7.5 Priorities

7.5.1 Priorities in atoms
The following example is composed of a single atom that can, at each state, either execute a transition labelled by the internal port `p`, or a transition labelled by the internal port `q`.

```cpp
package priorities_in_atom
port type Port()
atom type MyAtom()
port Port p(), q()
place LOOP
initial to LOOP
on p from LOOP to LOOP
```

7.7.3 Priorities in algorithms

```cpp
do { printf("2: free resource\n"); }
on sync from SYNC1, SYNC2 to GET1, GET2
do { printf("1 & 2: synchronize\n"); }
end
compound type HelloCompound()
component HelloAtom A()
end
```

Initially, both processes may acquire the resource since places `GET1`, `GET2`, `RESOURCE` are all marked initially. The, one of the two processes acquires the resource leading to a state in which place `RESOURCE` is not marked. This ensures the mutual exclusion between the use of the resource by the two processes: in this state, the other process cannot acquire the resource. Once the resource is released by a process it is blocked at place `SYNC1` or `SYNC2`, and the other process acquire, use and release the resource. Then both processes are in places `SYNC1` and `SYNC2` enabling the transition `sync` which leads to the initial state. An example of execution is provided below. Notice that we used the silent execution mode of the engine to remove debug information.

```bash
$ ./system --silent
1: get resource
1: free resource
2: get resource
2: free resource
1 & 2: synchronize
1: get resource
1: free resource
2: get resource
2: free resource
1 & 2: synchronize
2: get resource
2: free resource
1: get resource
1: free resource
1 & 2: synchronize
...```
compound type Layer3()
    component Layer2 L21(1), L22(5)
    connector Plus plus12(L21.ep, L22.ep)
    export port plus12.ep as ep
end
compound type HelloCompound()
    component Layer3 A12345678()
    connector Filter filter(A12345678.ep)
end

We provided for each exported port of compound the corresponding number of enabled interactions in the figure. When executing an instance HelloCompound we obtain the following execution sequence:

[BIP ENGINE]: initialize components...
[BIP ENGINE]: state #0: deadlock!

7.4 Petri nets

Most of the use cases of the BIP2 language consider automata for the behavior of atoms. In BIP2, it is also possible to use 1-safe Petri nets (see Petri net). The following BIP2 code is an example in which the behavior of an atom is a 1-safe Petri net representing concurrent accesses of two processes to a shared resource. States of the first (resp. second) process is represented by places GET1, USE1, SYNC1 (resp. GET2, USE2, SYNC2). The state of the resource is represented by place RESOURCE: its is marked whenever the resource is free.

Transitions represents actions of the system. With get1_res (resp. get2_res) the first (resp. second) process acquires the resource and use it (places USE1 or USE2). Transition free1_res (resp. free2_res) corresponds to the release of the resource by the first (resp. second) process. Transition sync synchronizes the processes and reset them to their initial states (places GET1 and GET2).

```cpp
@cpp(include="stdio.h")
package HelloPetriNet:
    extern function printf(string)
    port type Port()
    atom type HelloAtom()
    port Port get1_res(), get2_res(), free1_res(), free2_res(), sync()
    place GET1, GET2, RESOURCE, USE1, USE2, SYNC1, SYNC2
    initial to GET1, GET2, RESOURCE
    on get1_res from GET1, RESOURCE to USE1
do { printf("1: get resource\n"); }
on get2_res from GET2, RESOURCE to USE2
do { printf("2: get resource\n"); }
on free1_res from USE1 to SYNC1, RESOURCE
do { printf("1: free resource\n"); }
on free2_res from USE2 to SYNC2, RESOURCE
do { printf("2: free resource\n"); }
```

7.4. Petri nets 71
CHAPTER SIX
INSTALLING & USING AVAILABLE ENGINES

6.1 Requirements
The reference engine does not require any special software aside from a standard C++ compiler and the STL (usually installed along with the C++ compiler). In addition, a support for C++0x is required when working with the optimized engine, and C++11 for the multithread engine. We are currently working with the GNU compiler g++ version 4.8.2, and for ABI compatibility issues we recommend to use g++ version 4.8 or higher when compiling the generated code.

6.2 Downloading & installing

6.2.1 Getting latest version
Go to the download page for the BIP tools. As for the compiler, you may install the engines separately using specific archives, or you can install everything at once (compiler & engines). Only the first installation procedure is presented here. For the one archive installation, read the Downloading & installing.

6.2.2 Installation of the engine
The archive is a self-contained. You need to extract it in a dedicated directory, for example /home/a_user/local/bip2:

$ mkdir /home/a_user/local/bip2
$ cd /home/a_user/local/bip2
$ tar zxvf /path/to/the/BIP-reference-engine_2012.01.tar.gz
BIP-reference-engine-2012.01/
...

For easier use, set the following environment variables:
• BIP2_ENGINE_GENERIC_DIR: absolute path to generic header files.
• BIP2_ENGINE_SPECIFIC_DIR: absolute path to specific header files.
• BIP2_ENGINE_LIB_DIR: absolute path to library containing engine library.

This can be done by adding the following to your ~/.bashrc (if you are using bash):
export BIP2_ENGINE_SPECIFIC_DIR=/path/to/BIP-reference-engine-2012.01/include/specific
export BIP2_ENGINE_GENERIC_DIR=/path/to/BIP-reference-engine-2012.01/include/generic
export BIP2_ENGINE_LIB_DIR=/path/to/BIP-reference-engine-2012.01/lib/static

BIP2 Documentation, Release 2015.04 (RC7)
6.2.3 Quick-tour of installation

After extracting the archive, you should have a similar setup:

```
|-- generic/
|  |-- include
|     |-- AtomExportPortIf.hpp
|     |-- AtomInternalPortIf.hpp
|     |-- BipErrorIf.hpp
|  |-- specific/
|     |-- include
|     |  |-- AtomExportPort.hpp
|     |  |-- Atom.hpp
|     |  |-- AtomInternalPort.hpp
|     |  ...
```

- the `generic` directory contains the header files that should be common to all engines following the standard API (see dev-doc-engine_std_API-label).
- the `specific` directory contains the header files specific to the engine being installed (here, the reference engine).
- the `lib` directory contains the compiled code of the engine being installed.

6.3 Using the reference engine

6.3.1 Compiling & linking with generated code

You need to generate code from your BIP source as explained in More about C++ code generator.

Quick-start: follow regular cmake procedure

- create a `build` directory, for example within the generated code. This directory will host all files created during the compilation and the linking of the generated code. This directory can be wiped clean if needed without the need to run again the BIP compiler.
- from this new directory, invoke `cmake` by pointing to the directory containing the generated code.
- still from this new directory, invoke `make` to actually compile and link everything together

Step-by-step guide

You need to create a `build` subdirectory where all the compiled code will be located. Usually, this directory is a sub-directory within the generated code tree. For example, if the `output` directory contains all our generated code:

```
/home/a_user/output $ mkdir build && cd build
```

Then you need to invoke `cmake` from within this new `build` directory by pointing to the directory containing the generated code (in our example, ...). If you did not set environment variables as detailed in the Installation of the

7.3.2 Hierarchical components

The following example is a variant of the example of the previous section. We use a hierarchy of compounds instead of a hierarchy of connectors, but the principle remains the same.

```cpp
#include "stdio.h"

package HelloPackage

extern function printf(string, int, int)
port type HelloPort_t(int d)
atom type HelloAtom(int id)
data int active
export port HelloPort_t p(active)
place LOOP
initial to LOOP
do { active = 1; }
on p from LOOP to LOOP
provided (active == 1)
do { printf("I'm %d, active=%d\n", id, active); }
end
```

Notice that priorities—only maximal progress here—are applied globally to the hierarchical connector defined by the four layers of connectors. Enabled interactions of the connectors of the first, second and third layers are all taken into account, without applying maximal progress. The behavior would have been totally different if maximal progress was applying locally: in this case, only the interaction involving all the atoms would be enabled by the first layer, leading to a deadlock due to the guard of `filter`. This happens if the example if modified by structuring the system using compounds, as shown below.
6.3. Using the reference engine

Once the executable is built, help information is provided when executing the system.

The resulting executable is called `bip2_engine`.

6.3.2 Running the resulting executable

If your output matches the examples, you can proceed to the actual C++ compilation & linking by simply invoking:

```
$ cmake ..
```

Example cmake invocation without option `--help`:

```
$ cmake .. -DBIP2_ENGINE_LIB_DIR=/absolute/path/to/engines/BIP-reference-engine-2012.01/lib/static
```

Example cmake invocation with option `--help`:

```
$ cmake .. -DBIP2_ENGINE_LIB_DIR=/absolute/path/to/engines/BIP-reference-engine-2012.01/lib/static --help
```

The command `cmake ..` generates the building environment variables set:

```
-- Generating done
-- Configuring done
-- Detecting CXX compiler ABI info - done
-- Detecting CXX compiler ABI info
-- Check for working CXX compiler: /usr/bin/c++ -- works
-- Check for working CXX compiler: /usr/bin/c++
-- The CXX compiler identification is GNU
-- The C compiler identification is GNU
```

The command `cmake ..` generates the paths to engine files:

```
-- Check for working C compiler: /usr/bin/gcc -- works
-- Check for working C compiler: /usr/bin/gcc
-- The C compiler identification is GNU
-- The CXX compiler identification is GNU
```

For more information on the compiler options, you can search in the `BIP2-ENGINE-README.txt` file included in the engine.

If you import the `BIP2-ENGINE-EXAMPLES` directory, you can execute the code and compare your output with the examples.
BIP Engine general options:
- d, --debug: allows debug of the system, i.e. displays the state of the system
- e, --explore: execute a single sequence of interactions (default)
- i, --interactive: compute all possible sequences of interactions
- h, --help: display this help and exit
- l, --limit LIMIT: limits the execution to LIMIT interactions
- s, --seed SEED: set the seed for random to SEED
- v, --verbose: displays the display of the enabled/active interactions
- V, --version: displays engine version and exits

BIP Engine semantics options (WARNING: modify the official semantics of BIP!):
- --disable-maximal-progress: disable the application of maximal progress priorities
- --maximal-progress: enable the application of maximal progress priorities

Executing a single sequence

An execution sequence can be scheduled simply by running directly `system` without any option (execution of a single sequence is a default mode):

```
$ ./system
```

Notice that the reference engine is in verbose mode by default. At each state, it displays the enabled interactions and internal ports, and the chosen sequence, e.g.:

```
[BIPENGINE]: initialize components...
```

The behavior of instance of `HelloCompound` is as follows. The first layer of connectors enables interactions `A1.p`, `A2.p`, `A3.p`, `A4.p`, `A5.p`, `A6.p`, and `A7.p`, `A8.p`. These interactions are all visible from the exported port of the corresponding connectors. The second layer allows:

- any combination between interactions `A1.p`, `A2.p`, `A3.p`, `A4.p`, `A5.p`, `A6.p`, and `A7.p`, `A8.p`. Similarly, the third layer of connectors (i.e. `plus12345678`) allows any interaction between a subset of the height atoms (this corresponds to a total number of 255 interactions visible from the port `up` of `plus12345678`). We provided for each exported port of connector the corresponding number of enabled interactions in the figure. Notice that the value exported through this port for a given interaction corresponds exactly to the number of atoms involved in this interaction.

Due to the port defined in `filter`, the last layer of connectors limits the enabled interactions to the one that involve less than, or equals to, four atoms. The number of interactions enabled by `filter` is $162 = 70 + 56 + 28 + 8$, where 70 is the number of interactions involving 4 atoms, 56 is the number of interactions involving 3 atoms, 28 is the number of interactions involving two atoms, and 8 is the number of interactions involving only one atom.

The application of maximal progress to the enabled interactions of `filter` leads to only 70 maximal interactions which correspond to the interactions involving exactly four atoms. Once such an interaction is chosen an executed, the integer value associated to the port `up` of `plus12345678` is set to 0 by the function down of connector `filter`. This value is propagated recursively by down functions of connectors of type `Plus` to the variables active of the atoms involved in the executed interactions, and thus disabled their transition after their execution. As a result, there is only one maximal interaction at the next state of the model, which involves the four atoms that have not been executed by the previous execution of interaction. Its execution leads to a deadlock since all the atoms are inactive (i.e. active = 0 is false for all atoms).

A example of execution is provided below. It corresponds to the execution of `A1.p`, `A5.p`, `A7.p`, `A8.p` first, and then `A2.p`, `A3.p`, `A4.p`, `A6.p`. Notice that when atoms execute their transition, the value of active is 0 even if its value is 1 before executing. This comes from the fact that, in BIP, guards of atoms are tested at their stable states, that is, before synchronizing. The execution of an interaction may involve modification of the variables of the atoms due to down functions.

```
[BIPENGINE]: initialize components...
```

```
6.3. Using the reference engine

The current version of the reference engine requires an interaction on an atomic port to be executed before non-atomic components, which requires storing/retrieving their variables. For custom types, such code has to be provided.

Important:

- `--explore` option
- `--seed` option
- `--gencpp-enable-marshalling` option

Package HelloPackage

```cpp
@cpp(include="stdio.h")
```

The second layer connects the connectors of the first layer two by two, that is,

```cpp
connector type Plus(HelloPort_t r1, HelloPort_t r2)
```

down { r1.d = number_of_active; }
down { r2.d = number_of_active; }
end
```

Interactions or internal ports are chosen randomly amongst the enabled ones. The reference engine is based on a back-tracking, this mode of execution requires the generation of additional code, which is enforced using option `--gencpp-enable-marshalling`. The second layer connects the connectors of the first layer two by two, that is, the interactions `'AI.p'`, `'AJ.p'` and `'AI.p,AJ.p'`, and exports the number of atoms participating to the interaction through its exported port which connects the exported port of the connector of the previous layer (e.g., `plus1234.ep`), and exports the number of atoms participating to the interaction. The execution is stopped if no interaction and no internal port is enabled, or if `ctrl-D` is hit.

---

Choose a port or a port type to choose the interaction.

Example:

```cpp
export port HelloPort_t p(active)
data int active
data int number_of_active
```

```cpp
initial to LOOP do { active = 1; }
on p from LOOP to LOOP do { printf("I'm %d, active=%d\n", id, active); }
end
```

```cpp
connector type Filter(HelloPort_t r)
define r
```

```cpp
size_t custom_t_sizeof(const custom_t &v)
```

```cpp
• void custom_t_fromBytes(custom_t *ptr_v, const char *b)
• void custom_t_toBytes(char *b, const custom_t *ptr_v)
```

Imposes a guard that allows the interaction `'AI.p', 'AJ.p'` and exports the number of atoms participating to the interaction.

The number of bytes `n = custom_t_sizeof(*ptr_v)` must satisfy

- `n < number_of_active`
- `n >= 0`

The number of bytes provided (active == 1)

```cpp
do { printf("I'm %d, active=%d\n", id, active); }
```

The number of bytes that depend on the value of `v`

```cpp
on r1
on r1 r2
up { number_of_active = r1.d + r2.d; }
ep
```

The number of bytes which are stored in a location starting from `b`

```cpp
b = &custom_t
```

The execution is stopped if no interaction and no internal port is enabled, or if `ctrl-D` is hit.

---

The control version of the reference engine keeps an interaction on an atomic port to be executed before non-atomic components. The current version of the engine displays dots each time an interaction or an internal port is executed. Moreover, the number of reachable states, deadlocks, and errors is displayed,

```xml
found 27303 reachable states, 2 deadlocks, and 0 error in 0 state
```

...
6.4 Using the optimized engine

Since the reference engine (presented in the previous section) can be very, very slow, we recommend to use the optimized engine whenever performance is an issue. The optimized engine implements minimal optimizations required for reasonable runtime performances in terms of both execution time and memory usage. It currently passes the same tests as the reference engine, and it accepts the same general options.

For installing and using the optimized engine, proceed as explained above for the reference engine (see Installation of the engine), after downloading the optimized engine instead of the reference engine from download page. Performances can be again improved when combining the use of the optimized engine and the activation of optimizations in the code generator (see Optimization).

To allow maximal optimization, combine the following:

- pass --gencpp-opt-in 3 to the C++ back-end when compiling your BIP model
- use the optimized engine
- pass -DCMAKE_BUILD_TYPE=Release to cmake when compiling the generated C++ code (i.e. use cmake -DCMAKE_BUILD_TYPE=Release ..).

6.5 Using the multithread engine (beta version)

The multithread engine is proposed for increasing further the performance when running on multicore platforms. It is available in a beta version that is experimental in and should not be considered as mature as the reference and the optimized engine. It relies on the latest standard C++11 of C++, requiring version 4.8 or higher of GCC for compiling the generated C++ code. Moreover, it may require additional library implementing threads, e.g. to use pthread.

The options proposed by the multithread engine are listed below:

BIP Engine general options:
- -d, --debug (i.e. executes interactions in parallel, if obs. equivalent)
- -h, --help display this help and exit
- -i, --interactive interactive mode of execution
- -l, --limit LIMIT limits the execution to LIMIT interactions
- --seed SEED set the seed for random to SEED
- -s, --silent disables the display of the sequence of enabled/chosen interactions
- -v, --version displays engine version and exits
- --threads NB set the number of threads (by default, use the maximal HW parallelism or 8)
- -v, --verbose enables the display of the sequence of enabled/chosen interactions (default)

The multithread engine does not support any exploration mode and can only execute sequences of interactions. It executes components involved in interactions in parallel, based on the notion of partial state: interactions can start from partial states, that is, even if some components are still running. The multithread engine guarantees that interactions are always started in an order meeting the global state semantics which is implemented in the reference and the optimized engine.

Option --threads can be used to control the total number of threads used for executing the model. Notice that these threads are used not only for executing the atomic components, but also for computing the enabled interactions: connectors evaluate enabled interactions in a parallel and concurrent way.

Important:

This execution sequence also shows a interesting point about data handling. At the beginning, we can see:

```
ROOT.rcvrs.p4(1)d=135038644)
```

This value 135038644 shows that the corresponding data has never been initialized. Indeed, the compiler should have given you several warnings similar to this one:

```
[WARNING] In path/to/HelloPackage.bip:
'up' maybe missing: data associated with exported port won't be "fresh" ;
```

Please note that this is only a warning and not necessarily an error. As in this example, it can be completely valid to omit up() even with an exported port with data. As long as the entity bound to the exported port does not read port's data during the up(), there is no problem. The engine still displays the value of the data, which has no meaningful content.

Hint: As in almost every programming language, you should refrain from having uninitialized data: this practice is very error prone and often leads to hard to detect bugs.

7.3 Hierarchy in BIP2

7.3.1 Hierarchical connectors

The following example shows interesting aspects of the use hierarchical connectors. It is composed of height atoms A1, A2, A3, A4, A5, A6, A7, A8 that can execute only if they are active, that is, if their integer variable active equals to 1. They are initially active.

```
0
1
2
3
4
5
6
7
8

Figure 7.3: Structure of the model: 8 atoms, 4 levels of connectors (names of connectors of type Plus are not shown).
```

We consider four layers of connectors. The first layer connects atoms two by two with the connectors plus12, plus34, plus56, plus78 of type Plus. A connector plusIJ connects ports p of AI and AJ, and defines
The partial state semantics execution implemented by the multithread engine is equivalent to the one of the
global state semantics if the execution of components is side-effect free (i.e. the external code executed by a
component modifies only its local variables).

Due to the partial state semantics and the concurrent execution of connectors, the multithread engine cannot
 guarantee fairness of the execution of interactions and internal ports.

Notice that performances obtained when using the multithread engine depend on many factors, and may be worse than
the ones obtained when using the optimized engine. This is due to the overhead introduced by the use of threads and
threads synchronizations, which is inherent to the concurrent design implemented by the multithread engine.

6.6 Troubleshooting

6.6.1 libengine_path 
error when running cmake
If you get the following error:

CMake Error: The following variables are used in this project, but they are set to NOTFOUND.
Please set them or make sure they are set and tested correctly in the CMake files:

libengine_path

It's probably because you are trying to use a relative path for the
BIP2_ENGINE_LIB_DIR. Always use
absolute paths!

6.6.2 Atom.hpp: No such file or directory
error
If you get:

In file included from .../src/simple/AT_At1.cpp:3:
.../include/simple/AT_At1.hpp:6:20: error: Atom.hpp: No such file or directory

It's probably because you are trying to use a relative path for one or both
BIP2_ENGINE_GENERIC_DIR and
BIP2_ENGINE_SPECIFIC_DIR. Always use
absolute paths !

7.2.3 Wrapping components in a compound

Suppose we want to wrap the 3 receivers of the previous example into a single compound component, while keeping
the same global behavior. We simply need to build a compound component including the three receivers and the
connector that synchronizes them, and export the port of the connector at the interface:

```cpp
#include<stdio.h>
package HelloPackage
// [...] definitions of HelloPort_t, HelloSender, HelloReceiver,
// SyncReceivers and OneToOne
compound type RecvsCompound()
component HelloReceiver c1(1), c2(2), c3(3)
connector SyncRecvs sync(c1.p, c2.p, c3.p)
export port sync.ep as p
end

compound type HelloCompound()
component HelloSender s(0)
component RecvsCompound rcvrs()
connector OneToOne brd(s.p, rcvrs.p)
end

In this case, we obtain an equivalent execution sequence, that is:
...
[BIP ENGINE]: initialize components...
[BIP ENGINE]: state #0: 1 interaction:
[BIP ENGINE]: [0] ROOT.brd: ROOT.s.p({d}=0;) ROOT.rcvrs.p({d}=135034644;)
[BIP ENGINE]: -> choose [0] ROOT.brd: ROOT.s.p({d}=0;) ROOT.rcvrs.p({d}=135034644;)
I'm 0, sending Hello World....
I'm 1, Hello World received from 0
I'm 3, Hello World received from 0
[BIP ENGINE]: state #1: deadlock!
```

Figure 7.2: Structure of an instance of HelloCompound.

Notice that in the above example, only maximal interactions of
sync are visible from
brd, since priorities are applied
to exported port of compounds. The resulting behavior is equivalent to the one obtained when using a hierarchical
connector without encapsulating the receivers in a compound, but this is not the case in general, as explained as
64 Chapter 7. Tutorial
Figure 7.1: Broadcast from $s$ using a single connector (left) or a hierarchical connector (right).

```cpp
#include "stdio.h"

package HelloPackage

// [... definitions of HelloPort_t, HelloSender and HelloReceiver [...]

connector type SyncRecvs(HelloPort_t r1, HelloPort_t r2, HelloPort_t r3)
    data int d
    export port HelloPort_t ep(d)
    define r1' r2' r3'
    on r1 r2 r3 down { r1.d = d; r2.d = d; r3.d = d; }
    on r1 r2 down { r1.d = d; r2.d = d; r3.d = d; }
    on r2 r3 down { r2.d = d; r3.d = d; }
    on r1 down { r1.d = d; }
    on r2 down { r2.d = d; }
    on r3 down { r3.d = d; }
end

connector type OneToOne(HelloPort_t s, HelloPort_t c)
    define s' c
    on s c down { c.d = s.d; }
end

compound type HelloCompound()
    component HelloSender s(0)
    component HelloReceiver r1(1), r2(2), r3(3)
    connector SyncRecvs sync(r1.p, r2.p, r3.p)
    connector OneToOne brd(s.p, sync.ep)
end

The computation of the interactions in the hierarchical connector composed of $brd$ and $sync$ is as follows. First, all the enabled interactions of $sync$ are computed, that is, '$r1', '$r3', and '$r1,r3'$. Then, from these interactions the enabled interactions of $brd$ are computed leading to the following enabled interactions for $brd$: '$s', '$s,r1', '$s,r3', and '$s,r1,r3'$. The application of priorities (i.e. maximal progress) to the enabled interactions of $brd$ leads to the following execution:

...
HelloReceiver initial transitions are the following: 

Due to the guard of the transition labelled by \texttt{sync}, it is still considered as a possible interaction, but no transfer of data occurs when \ldots

To implement the broadcast of data from port \texttt{s}, we use a list of \texttt{s}, \texttt{s,r1}, \ldots, \texttt{s,r1,r2,r3}. As explained in \ldots

Priorities of maximal progress (the default priority rules of BIP2) leads to the execution of the maximal interaction \ldots


defined by \texttt{p} from \texttt{START} to \texttt{END}

place \texttt{START,END}

initial to \texttt{START}

\texttt{port HelloPort_t p()}

on \texttt{p} from \texttt{START} to \texttt{END}

\texttt{place \texttt{START,END}}

\texttt{export port HelloPort_t p(myd)}

\texttt{connector OneToThree brd(s.p, r1.p, r2.p, r3.p)}

\texttt{component HelloSender s(0)}

\texttt{component HelloReceiver r1(1), r2(2), r3(3)}

\texttt{on s r3 down \{ r3.d = s.d; \}}

\texttt{on s r2 down \{ r2.d = s.d; \}}

\texttt{on s r1 down \{ r1.d = s.d; \}}

\texttt{on s r2 r3 down \{ r2.d = s.d; r3.d = s.d; \}}

\texttt{on s r1 r3 down \{ r1.d = s.d; r3.d = s.d; \}}

\texttt{on s r1 r2 down \{ r1.d = s.d; r2.d = s.d; \}}

\texttt{on s r1 r2 r3 down \{ r1.d = s.d; r2.d = s.d; r3.d = s.d; \}}

\texttt{define s' r1 r2 r3}

\texttt{\doit{\{ printf("I'm %d, Hello World received from %d\n", id, myd); \}}}}

\texttt{provided (id == 1 || id == 3)}

The package contains 3 types:

\texttt{\[BIP ENGINE\]: state \#0: 1 interaction:}}

\texttt{end}

\texttt{\[BIP ENGINE\]: initialize components...}}

\texttt{...}

\texttt{\[BIP ENGINE\]: \[0\] ROOT.brd: ROOT.s.p({d}=0;)}

\texttt{ROOT.r1.p({d}=0;)}

\texttt{ROOT.r3.p({d}=0;)}

\texttt{\[BIP ENGINE\]: state \#1: deadlock!}}

\texttt{I'm 3, Hello World received from 0}

\texttt{I'm 1, Hello World received from 0}

\texttt{I'm 0, sending Hello World....}

\texttt{\[BIP ENGINE\]: \rightarrow choose [0] ROOT.brd: ROOT.s.p({d}=0;)}

\texttt{ROOT.r1.p({d}=0;)}

\texttt{ROOT.r3.p({d}=0;)}

\texttt{\[BIP ENGINE\]: \[0\] ROOT.brd: ROOT.s.p({d}=0;)}

\texttt{ROOT.r1.p({d}=0;)}

\texttt{ROOT.r3.p({d}=0;)}

\texttt{\[BIP ENGINE\]: \[0\] ROOT.brd: ROOT.s.p({d}=0;)}

\texttt{ROOT.r1.p({d}=0;)}

\texttt{ROOT.r3.p({d}=0;)}

\texttt{\[BIP ENGINE\]: \[0\] ROOT.brd: ROOT.s.p({d}=0;)}

\texttt{ROOT.r1.p({d}=0;)}

\texttt{ROOT.r3.p({d}=0;)}

\texttt{\[BIP ENGINE\]: \[0\] ROOT.brd: ROOT.s.p({d}=0;)}

\texttt{ROOT.r1.p({d}=0;)}

\texttt{ROOT.r3.p({d}=0;)}

\texttt{\[BIP ENGINE\]: \[0\] ROOT.brd: ROOT.s.p({d}=0;)}

\texttt{ROOT.r1.p({d}=0;)}

\texttt{ROOT.r3.p({d}=0;)}

\texttt{\[BIP ENGINE\]: \[0\] ROOT.brd: ROOT.s.p({d}=0;)}

\texttt{ROOT.r1.p({d}=0;)}

\texttt{ROOT.r3.p({d}=0;)}

\texttt{\[BIP ENGINE\]: \[0\] ROOT.brd: ROOT.s.p({d}=0;)}

\texttt{ROOT.r1.p({d}=0;)}

\texttt{ROOT.r3.p({d}=0;)}

\texttt{\[BIP ENGINE\]: \[0\] ROOT.brd: ROOT.s.p({d}=0;)}

\texttt{ROOT.r1.p({d}=0;)}

\texttt{ROOT.r3.p({d}=0;)}

\texttt{\[BIP ENGINE\]: \[0\] ROOT.brd: ROOT.s.p({d}=0;)}

\texttt{ROOT.r1.p({d}=0;)}

\texttt{ROOT.r3.p({d}=0;)}

\texttt{\[BIP ENGINE\]: \[0\] ROOT.brd: ROOT.s.p({d}=0;)}

\texttt{ROOT.r1.p({d}=0;)}

\texttt{ROOT.r3.p({d}=0;)}

\texttt{\[BIP ENGINE\]: \[0\] ROOT.brd: ROOT.s.p({d}=0;)}

\texttt{ROOT.r1.p({d}=0;)}

\texttt{ROOT.r3.p({d}=0;)}

\texttt{\[BIP ENGINE\]: \[0\] ROOT.brd: ROOT.s.p({d}=0;)}

\texttt{ROOT.r1.p({d}=0;)}

\texttt{ROOT.r3.p({d}=0;)}

\texttt{\[BIP ENGINE\]: \[0\] ROOT.brd: ROOT.s.p({d}=0;)}

\texttt{ROOT.r1.p({d}=0;)}

\texttt{ROOT.r3.p({d}=0;)}

\texttt{\[BIP ENGINE\]: \[0\] ROOT.brd: ROOT.s.p({d}=0;)}

\texttt{ROOT.r1.p({d}=0;)}

\texttt{ROOT.r3.p({d}=0;)}

\texttt{\[BIP ENGINE\]: \[0\] ROOT.brd: ROOT.s.p({d}=0;)}

\texttt{ROOT.r1.p({d}=0;)}

\texttt{ROOT.r3.p({d}=0;)}

\texttt{\[BIP ENGINE\]: \[0\] ROOT.brd: ROOT.s.p({d}=0;)}

\texttt{ROOT.r1.p({d}=0;)}

\texttt{ROOT.r3.p({d}=0;)}

\texttt{\[BIP ENGINE\]: \[0\] ROOT.brd: ROOT.s.p({d}=0;)}

\texttt{ROOT.r1.p({d}=0;)}

\texttt{ROOT.r3.p({d}=0;)}

\texttt{\[BIP ENGINE\]: \[0\] ROOT.brd: ROOT.s.p({d}=0;)}

\texttt{ROOT.r1.p({d}=0;)}
The expected behavior, when considering a system with a component of type HelloCompound as the root, is a deadlock after the only transition labelled by p is executed in the atom c1.

For the sake of the example, we want to show an execution of this model and thus we use the C++ back-end along with the reference engine. But this is not mandatory (but as of this writing, it’s the only option to execute BIP).

Compile it using the following commands for producing C++ code that is compiled and linked with the reference engine:

```
$ mkdir output
$ bipc.sh -I . -p HelloPackage -d "HelloCompound()"
--gencpp-output output
$ mkdir output/build
$ cd output/build
$ cmake ..
[...]
$ make
[...]
```

And finally, run the produced system executable:

```
$ ./system
[...]
[BIP ENGINE]: initialize components...
[BIP ENGINE]: state #0: 1 interaction:
[BIP ENGINE]: [0] ROOT.connect: ROOT.c1.p() ROOT.c2.p() ROOT.c3.p()
[BIP ENGINE]: -> choose [0] ROOT.connect: ROOT.c1.p() ROOT.c2.p() ROOT.c3.p()
Hello World from 1
Hello World from 2
Hello World from 3
[BIP ENGINE]: state #1: deadlock!
```

After the only transition is triggered, the system reaches a deadlock state, as expected.

### 7.2 Synchronizing components using interactions of BIP2

#### 7.2.1 Rendez-vous between several components

We modify the example of Section Hello world so that we now have three instances of the atom type Hello Atom instead of only one, and we force them to synchronize their single transition (i.e., the rendez-vous):

@cpp(include="stdio.h")
package HelloPackage

extern function printf(string, int)

port type HelloPort_t()

atom type HelloAtom(int id)
export port HelloPort_t p()

place START, END
initial to START
on p from START to END do {printf("Hello World from %d\n", id);}
end

connector type ThreeRendezVous(HelloPort_t p1, HelloPort_t p2, HelloPort_t p3)
define p1 p2 p3
end

compound type HelloCompound()

- component HelloAtom c1(1), c2(2), c3(3)

The annotation @cpp() is explained later on and allows us to use the printf() from the C standard library. In this example, we add a connector type ThreeRendezVous with three port parameters of type HelloPort_t. It defines exactly one interaction that synchronizes the three ports.

Compile it using the following commands to produce C++ code that is compiled and linked with the reference engine:

```
$ bipc.sh -I . -p HelloPackage -d "HelloCompound()"
--gencpp-output output
$ mkdir output/build
$ cd output/build
$ cmake ..
[...]
$ make
[...]
```

When running the executable, you can see that the transitions of the three atoms are triggered simultaneously. The execution of the three atoms is sequentialized in an arbitrary order, e.g.:

```
[...]
[BIP ENGINE]: initialize components...
[BIP ENGINE]: state #0: 1 interaction:
[BIP ENGINE]: [0] ROOT.connect: ROOT.c1.p() ROOT.c2.p() ROOT.c3.p()
[BIP ENGINE]: -> choose [0] ROOT.connect: ROOT.c1.p() ROOT.c2.p() ROOT.c3.p()
Hello World from 1
Hello World from 2
Hello World from 3
[BIP ENGINE]: state #1: deadlock!
```

#### 7.2.2 Broadcasting data to several components

We now consider an example composed of one component—the sender—that broadcasts an integer variable representing its identifier to three other components, the receivers. The corresponding BIP2 code is the following.

@cpp(include="stdio.h")
package HelloPackage

extern function printf(string, int)
extern function printf(string, int, int)

port type HelloPort_t(int d)

atom type HelloSender(int id)
data int myd
export port HelloPort_t p(myd)

place START, END
initial to START
on p from START to END do {printf("I'm %d, sending Hello World...\n", myd);}
end

connector type ThreeRendezVous connect(p1, p2, p3)
end
end

atom type HelloReceiver(int id)
data int myd