BIP2 Documentation

Release 2015.04 (RC7)

VERIMAG

April 30, 2015

CONTENTS

CHAPTER

ELEVEN

INDICES AND TABLES

- *genindex*
- *search*

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/cli/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.

BIP2 Documentation, Release 2015.04 (RC7)

Contents:

Do not forget the leading `.` in the command above.

10.2 Publishing a distribution

Publishing a distribution must always includes the following steps:

1. **compute a new version name**. You must never reuse 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 (*eg.* 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 Never provide outdated documentation. If you can't update the documentation (it's a shame), outdated part must be explicitly marked.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 (*ie.* 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:

```
# The short X.Y version.
version = '2012.04'
# The full version, including alpha/beta/rc tags.
release = '2012.04 (RC3)'
```

Building is as easy as running:

```
$ 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:

```
$ svn tag version-name
```

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.

10.1.2 What `wrap.sh` does

It starts by cleaning everything : its own `build` directory along with the separate build directories for the difference parts of the compiler. And immediately starts the building of all parts of the compilers. It does that by using `ant` with the following targets:

- `clean`: to remove previous build artefacts
- `publish-local-all`: build the compiler

The `publish-local-all` target is a frontend target that uses `ivy`.

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.

10.1.3 What `wrap.sh` produces

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

```
build
+-- 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.

10.1.4 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
```


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 command 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
```

CHAPTER

TEN

DEVELOPPER 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 (*ie.* 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**.

CHAPTER

TWO

THE BIP2 LANGUAGE

2.1 Introduction

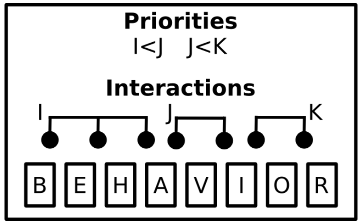


Figure 2.1: The three-layered BIP2 representation.

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

- the *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* that corresponds to 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* components 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 then 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)* ')']

annotation_parameter :=
    annotation_key
  | annotation_key '=' annotation_value
  | annotation_key '=' '"' annotation_string_value '"'
```

Example

```
@cpp(foo=bar, obj="foo.o,bar.o")
atom type MyAtom(int x)
 ...
end
```


Middleends & Backends modifications

After the previous changes are working (you can run the compiler without executing any middlend/backend to check that everything is fine), you can proceed with their modifications.

9.8.4 Updating dependencies

You should always try to stick to latest stable version of all dependencies. Not doing so may lead to big problems when you will try to update from very old libraries. It's easier to fix little API change from one version to the next than fixing a large set of changes.

Usually, you should simply follow these simple steps. Beware that sometimes, some dependencies must be added or can be removed. For a given compiler module:

- list jar files located in the ''external/" directory
- check if you have more recent version of these jar files inside your ''plugins/" subdirectory in your eclipse installation.
- for all jar with more recent version, replace old version by the newest version
- run the ''Tools/helper-scripts/gen-ivy-deps2.py'':

  ```
  externals/ $ ls *.jar | Tools/helper-scripts/gen-ivy-deps2.py
  ```

- copy the result inside the ''ivy.xml" file (first remove the old dependencies related to jar files)

That's it. **Always** run all the test before merging. It is very important, as it has happened that some class moved from one jar to another and the compiler crashes in very specific cases.

- implement the check. If you need to create error or warning, do the following:
  - choose between *error* or *warning* by changer (or leaving) the `Diagnostic.ERROR` as the first parameter of the `createDiagnostic()` method call
  - the 6th parameter is an array of `Object`. Add the error code corresponding to the error/warning in second position of this new array.
- commit the handwritten code and add `[MODEL CODE MODIF]` in the commit message and give as much info as you can (which constraint, which ticket, ...)
- implement a new unit test

You can see this kind of hack in commits r5062 though r5068 of the BIP2 subversion repository.

9.8.3 Changing the syntax

A change in the syntax can impact the compiler in various ways:

- only the first parser pass: most likely, the change is syntatic-suggar related. The change is invisible after the first parser (*ie.* the AST structure given to the walker is unchanged).
- only the parser: the AST produced is different, and the walker needs to be adapted as well, but the resulting model still uses the same metamodel (*ie.* still dealing with syntactic suggar).
- parser and meta-model are impacted: this means that you need to change the grammar, the walker, the meta-model and also all middleends & backends.

Important: Do not forget to add corresponding tests in the unit test database! Run theses tests as much as possible to check that you are not breaking something.

Grammar modification

Change the ''Bip2.g" to match your syntax change. If your changes do not change the kind of AST the parser produces, it's really easy and quick: you're done (run the ''gencode-for-eclipse" ant target if you need these changes to be visible in your eclipse).

If your changes DO change the produced AST, you need to add the *imaginary nodes* at the top of the ''Bip2.g" file and proceed with the next section.

Walker modification

If you changed the AST produced by the first pass or if you need to change the model produced, you need to modify ''Bip2Walker.g''.

You need to take *extra* care about the asumptions (often implicit) made at the interface between the 2 passes. Some abstractions can be safely made in the walker, but your change may change this: be extra-careful. For example, in the first pass, the rules will forbid some expressions in some context. No indirect data reference can be made in an atom transition. But this restriction does not exist in the walker: the walker trusts the first pass.

Meta-model modification

If you need to change the meta-model, you should make these changes *before* changing the walker (you won't be able to change the walker before...).

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)* ] ')'
```

Example

```
package SomePackage
  const data int my_const_int = 42

  extern data type my_list

  extern function int min(int, int)
  extern function printf(string)
  extern function display(my_list)
  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:

```
data int x
```

declares a variable named `x` of type `int`. The keyword `data` is ommited 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`)
- `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.

The templates `startPackageCmake`/`endPackageCmake` must be called respectively at the beginning/ending of a package.

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 folowing 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` than `BadType`.
- commit the change in the ecore file.
- generate the code. This will create an empty method with a `FIXME` inside (look in the `Tasks` perspective in eclipse, the new method should appear here).
  - open the `genmodel` file, right click on `Bip2` and run the *Generate model code*
- commit the generated code corresponding to the new constraint. In the comment, add explicitly that it is only automatic code
- add the corresponding error message and error code in the `u.v.b.error` module:
  - add a new enumeration item in the `ErrorCodeEnum` Enum type: item name must match the constraint name
  - if needed, add one more mapping in the ''ErrorCodeEnum.userFriendlyNames" map to map your newly *verbose* name to some shorter names.
  - add a new string in the error message file `english-strings.properties`. The new string name must match the constraint name
- implement the constraint.
  - add **NOT** after the *@generated* in the comment before the method.

Important: The *limitation* only exists for simplification purposes. It is completely possible to compile only BIP types into C++ classes and package the result as a library, but our current compilation flow does not support the use of precompiled BIP package. This feature will be handled in later version.

9.7.1 Type code generation

All type templates (*ie.* template for any sub-class of the `BipType` class in the meta-model) must conform to the following interface (not conforming templates won't raise any compilation error, but will most certainly produce wrong code in an unspecified manner), with `XXXXType` the sub-class name. The interface is defined in `generateBipType.mtl`:

- `generateHeaderBody(anElt: XXXXType, disableSerialization : Boolean)`, **mandatory**. The content of the *main* header file. No need to handle the multiple-inclusion guards. Always include only the minimum set of files: never use the *include everything as it's easier to implement* strategy as you'll quickly introduce loops.
- `generateImplemBody(anElt : XXXXType, disableSerialization: Boolean)`, **mandatory**. The content of the cpp file. No need to include the corresponding hpp. Only implement class's members or **static** functions. Avoid non-static functions as it *violates* the design principles.
- `generateSubClasses(anElt : XXXXType, aCMakeList : String, disableSerialization: Boolean)`, **optional**. When the generation process needs to produce more than 1 class for a given BIP type, you need to hook your *other templates* in this template. The `aCMakeList` is the filename to use to append cmake instructions relative to the *other classes* produced.

The `disableSerialization` parameter can be set *true* when all serialization mechanisms should be skipped. This parameter should be moved to some higher global context instead of being part of all template interfaces.

Examples of templates using the sub-class generation include: `generatePortType.mtl` and `generateConnectorType.mtl`.

9.7.2 Instance deployment code generation

The `generateDeploy.mtl` template is responsible for walking the instance model, that can be seen as a tree if you omit type references that point to the type model. It uses the recursive aspect of the component hierarchy to *unroll* the instance tree and create C++ object declaration in an order that meets all the classes constructors requirements (*eg.* a compound constructor expects references to all its connectors, priorities, components, exported data and exported ports). As much as possible, the template uses static initializations to minimize runtime initializations and allow for better optimization from the C++ compiler. It means that there is no `new` calls in the generated code for deployment: the size of the system can be statically known after the C++ compiler is done (it does not include, of course, runtime data like interactions objects) and everything is allocated in the heap.

The generated code includes a `Component* deploy(int argc, char **argv);` function that is the entry point that standard engine use. Currently, `argc` and `argv` are not used. This function returns a pointer to the root component instance.

9.7.3 CMake

The template `generateMasterCMakeLists` produces the main `CMakeLists.txt` file that will be used to configure and compile all C++ code produced by the compiler (*ie.* packages and deployment). It expects a set of parameters from the user interface. Some of these parameters are directory lists and should be given as absolute path. Using relative path may or may not work depending on the specific setup: it is *not* supported and should not be used.


```
extern data type IntList refine List as "std::list<int>"
```

This declaration states that `IntList` is a valid type name. It also specifies that `IntList` is a sub-type of the (external) type `List`, and that `IntList` should be translated into `std::list<int>` by code generators (e.g. in this example we target C++ code generation). Code generators use the name of the type (in this example `IntList`) if the instruction `as` is not provided, e.g. when using the following declaration code generators will not translate `IntList` and use its name directly in the generated code.

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

Important: Without any additional declation, the compiler assumes that no operation can be performed on external types except assignments (using `=`). This means that assignments of external types should be implemented in the generated code, e.g. by additional files included in the compilation process.

As for external types, BIP2 allows the declaration of external function prototypes that are assumed to be externally defined and present when compiling the generated code. The declaration of an external function consists of an optional return type name, a function name, and a list of types names for the arguments of the function. For example:

```
extern function int rand()
extern function printf(string)
extern function int getElement(int, IntList)
```

declares prototypes for:

- the external function `rand` having no argument and returning an `int`
- the external function `printf` that takes a `string` as argument and have no returned value
- the external function `getElement` that takes an `int` and an `IntList` as arguments, and returns an `int`.

Important: External function prototypes may involved external data types (that must be declared properly obviously). There are no specific restrictions in the declaration of prototypes concerning overloading: different prototypes may have the same function name even if they have the same number of arguments and/or different return types. This may trigger errors when compiling expressions involving calls to external functions, as explained in *Actions*.

2.2.5 Actions

Actions define computations and data transformations. In the *constant* context, expressions should not have side effects. Notice that the compiler is unable to check whether an external function involved in a constant context has side effects. It is the user's responsibility to ensure the absence of side effects in such context. In the *non-constant* context any computation is allowed. There are also *mixed* contexts where some data can be changed while others can't (see *Connectors*). Whenever possible, the compiler will restrict the possible actions to enforce the "const-ness".

Computations and data transformations in actions are expressed by C-like syntax statements and expressions. Statements are assignments, function calls and conditional `if-then-else` constructs. Notice that the language has no support for loops. Expressions involved in statements can combine values using comparison operators, arithmetics operators, boolean operators, and function calls (with returned values). As usual, parenthesis `(` and `)` may be used to group expressions and enforce a specific evaluation order. Multiple statements in an action are enclosed in brackets while individual statements are separated by `;`. The following operators can be used for native types.

Comparison operators can be used to compare two values of the same native type and return a value of type `bool`. In addition we also allow the comparison of `int` to `float` and `float` to `int`. The list of comparison operators is provided as follows.

- `==` : equality
- `!=` : inequality

- `<` : *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*

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 (*ie.* 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* (*ie.* creates instances of previously created classes, in a correct order).

- an example

Hint: The *pipe* like syntax used to chain the filters from the command line is handled in the command line user interface, not in this module.

9.6 Back-end

9.6.1 `u.v.b.backend`

This module contains 3 elements shared by all the back-ends and needed for interacting with the other parts of the compiler:

- the `Backendable` interface that back-ends must implement.
- the acceleo runtime, that is meant to be used by all back-ends (even though a back-end can be in pure Java)
- some acceleo template/queries that are useful for all back-ends (*eg.* extracting information from annotations, some other common operations, ...)

Important: As of this writing, acceleo has some limitation (bug) that prevent the real sharing of common templates/queries. The templates/queries provided here are currently copied in all back-ends modules that need them. This is a work-around.

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

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

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

This back-end produces BIP code. It is very simple, as templates are used to translate the *BIP-EMF* to the textual BIP representation, with both being by construction very close.

There are 6 templates:

- 4 BIP types (*ie.* port, connector, atom, compound)
- 1 for the package
- 1 for the port declarations
- 1 for the annotations

This backend can be a good starting point for understanding the internals of the backends using acceleo templates.

Warning: When writing unit test for BIP, we mainly use the EMF *equals()* method to check that *bip(bip(a-test-source))* == *bip(a-test-source)*. EMF models are sensitive to *order*, meaning that even if some model are equivalent from the BIP point of view, they are not from EMF point of view. For example, the generated code will *always* have : data types, port types, connector types, atom types, compound types. Same goes for atom internals, where data comes before export port, that comes before internal ports.

- it matches at least two prototypes without one beging strictly more precise than the other one (“ambiguous function call” error)
- the return type of the most precise matching prototype is not compatible with the rest of the expression in which the function is called (“incorrect type” error)
- the most precise matching prototype has no return type and the function call is involved in an expression (“no return value” error).

Considering that the prototypes for `min` are restricted to the following:

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

then the statement `x = min(0, 0);` will lead to a compilation error such as:

```
[SEVERE] In /path/to/file/my_bip_file.bip:
Ambiguous function call 'min' with parameter(s) of type(s) 'int, int': cannot decide
between 'float min(float, int), float min(int, float)' :
    38:
    39:         x = min(0, 0);
--------------------^
    40:
    41:
```

Similarly to external functions, external operators can be declared by using `extern operator` followed a return type, the target operator (instead of the function name) and its arguments, e.g.:

```
extern operator string +(string, string)
```

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 whereas the last two ones are accepted:

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

Notice that declarations of external comparison operators (==, !=, <, >, <=, >=) are not forced to return boolean values, but for readability of the code we recommend to avoid such practice. Similarly, logical operators (`!`, `||`, `&&`) may be redefined for non boolean values, but again we strongly recommend not doing it:

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

Example

```
{
  a = a * (2 + b);
  g(d);
  b = f(a);
}
```

In a constant context, an action contains a single expressions enclosed in parenthesis that must evaluate to a boolean value.

Important: Depending on the locations of the actions, the data reference can take different forms. For example, 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:

```
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:

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

Syntax

- **accepts annotations**

```
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:

```
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 ''@SuppressWarning'' 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.

`instantiate(*Declaration declaration)` methods while browsing the types found for all sub-declarations (*eg.* taking an instance of a compound triggers the instantiation of sub-components, connectors, priorities, ports).

More details on the instantiation of component

The entry point of the instantiator is the the ''instantiateTopLevel(ComponentDeclaration declaration)" method. The component declaration must be a compound, else it will fail. This method will simply *unroll* the hierarchy starting from the root compound and build an instance tree. Each encountered declaration (port, data, connector, component, priority, etc) will trigger the creation of an instance object in the tree (the instance objects make a tree). It is important to note that components parameters need special handling.

Parameters for a component declaration can only involve the following:

- direct values: 3, 18.5
- data references to container's data parameters
- data references to constant data declaration

There is a need when instantiating a parameterized component declaration to resolve the data references, in particular for reference to container's parameters. What we do is that we duplicate the expressions found in the declaration (involving only objects within the type graph) and then we resolve data references to point to instance objects instead of pointing to type objects.

9.3.4 `u.v.b.userinterface.cli`

Any user interface is expected to instantiate compiler's building blocks and assemble them to create a working compiler. This module contains a *command line interface*.

It basically does the following steps:

- initializes the command line parsing tool:
- with common arguments (package, verbosity, search paths, root declaration, etc)
- with arguments from back-ends (this is achieved by introspecting the back-ends classes)
- creates a package loader
- loads the package requested from the command line
- if a root declaration was given, instantiate it
- executes all back-ends in turn.

All steps may fail and should report the cause by transmitting an Error object. The actual class and mean of transmission depends on the step failing.

9.3.5 `u.v.b.frontend.tests`

Tests are using JUnit and follow the conventions:

- classes with tests are named `SomethingTests`, with `Something` being explicit enough about the content. The class name can't start with `Abstract`.
- tests that need the package loader are located in the package `loader`. Store other tests in separate packages: keep tests tidy!

Ports

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)
  ...
end
```

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.

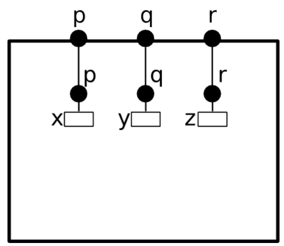


Figure 2.2: Ports `p`, `q` and `r` are individually exported.

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

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

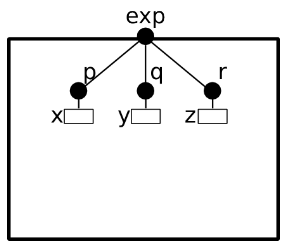


Figure 2.3: Ports `p`, `q` and `r` are merged and exported as the port `exp`.

To merge and export ports `p`, `q` and `r` as a single port `exp` we use the keyword `as`:

```
export port 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`.
- The set of *target* places that are marked after its execution. They are declared using the keyword `to`.
- 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`:

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 work-flow:

- `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 (*eg.* 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 developement.

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.instantiator`

The instantiator 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 `*Instance` java classes 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 `*Instance`

```
                  theElementWithAConstraint,
                }
                context));
```

You must add an extra information to identify precisely the exact error detected by the constraint (codes are defined in `ujf.verimag.bip.error` module : *u.v.b.error*). This code must be added in the array of `Object` created near the end of the previous excerpt:

```
diagnostics.add(createDiagnostic(Diagnostic.ERROR,
                  DIAGNOSTIC_SOURCE, 0,
                  "_UI_GenericConstraint_diagnostic",
                  new Object[] {
                    "exportedDataListsSynchronized",
                    getObjectLabel(theElementWithAConstraint, context)
                  },
                  new Object[] {
                    theElementWithAConstraint,
                    /*
                     * BIP Error code corresponding to this constraint
                     */
                    ErrorCodeEnum.constraintXYViolation,
                  },
                  context));
```

EMF allows the use of different *level* for each `Diagnostic` object created. In the BIP compiler, we only use the `ERROR` when the constraint violation is fatal (*ie.* the compiler must stop) and `WARNING` when the violation is a sign of potential error (in general, these can only be detected at runtime).

Versionning generated code

As we need to implement our constraints in the generated code, we need to add it in the code repository. In order to differentiate handwritten code (that we really need to keep track of) from automatically generated code (thousands of lines), split your commits ! Use the following steps :

- add constraints in the meta-model
  - commit changes in `.ecore` and `.ecorediag` files with regular comments.
- generate java code
  - commit only the generated code and state that this is generated code for new constraints
- implement the constraints
  - commit your changes with `[MODEL CODE MODIF]` with the names of the constraints you've modified.

Important: When adding a new constraint, *always* **always** create the corresponding error message and JUnit test at the same time. *Never* commit the constraint code if you don't have the tests and errors ready. If you do so, you *will* forget about them and hit problems later. See corresponding sections for adding error and tests.

9.3.2 `u.v.b.parser`

For historical reasons (*ie.* no real technical reasons), the `parser` modules contains not only a parser, but also the code for the *package loader* and its *package registry*.


```
on s from SYNC to END
```

The following figure gives an example of execution sequence of transitions in an atom `A` in which the initial transition is followed by the execution of an internal transition, then a transition labeled by port `p` is executed followed by the execution of two internal transitions, and finally a transition labeled by port `q` is executed leading to a state in which no transition is enabled. Notice that the only visible states of `A` are the ones preceding the executions of `p` and `q` and the final state, the other intermediate states are invisible.

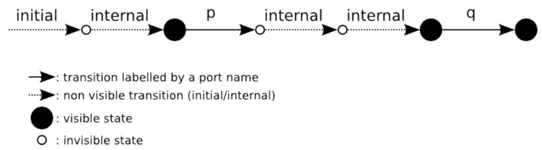


Figure 2.4: Sequence of internal and visible transitions in an atom.

Important: Only one internal transition is enabled at any time since non-determinism is not allowed for internal transitions of atoms. Similarly, two transitions labeled by the same internal port name must not be enabled at the same time.

Priorities

Priorities are used to resolve conflicts or to define an ordering between transitions labelled by ports: the selected transition corresponds to the port with highest priority. They may also include a boolean expression called *guard* that specifies the conditions when it is applicable. Priorities do not apply to the initial and internal transitions. In the following example, port `q` has higher priority than `p` provided that variable `x` equals to zero.:

```
priority myPrio p < q  provided (x==0)
```

The transitive closure of such priorities defined in an atom is a partial order relationship among ports and associated transitions. A port `q` has higher priority than `p` if there is a priority rule specifying `p < q` whose guard evaluates to true, or there are ports `p1`, `p2`, ..., `pN` such as `p < p1 < p2 < ... < pN < q` such that all their corresponding guards evaluates to true. Notice that it is not required for ports `p1` to `pN` to be enabled. An enabled transition is *maximal* if it has the highest priority.

Important: Inconsistencies in priorities (e.g. `a < b < c < a`) are detected and reported. If the priorities do not include guards, the checks are performed at compile time. Guard expressions can not be evaluated during the model compilation so in this case priority validation is postponed until run time.

Enabled ports of atoms

An internal port is *enabled* if it triggers an enabled transition for the current atom state. The port is *maximal* if its corresponding enabled transition is maximal. An exported maximal port is also enabled at the interface level. When 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.

The following figure illustrates an example of a merged port named `exp` that consists of three internal ports `p`, `q` and `r` and each internal port references a variable (e.g. `x`, `y` and `z`). Port `exp` is enabled if at least one of the corresponding ports is enabled. However, only the variables of the enabled internal ports are accessible from the interface. For 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.

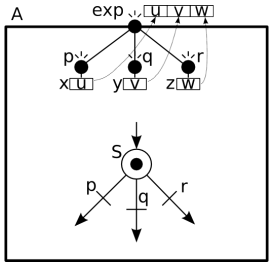


Figure 2.5: Example of a port enabled by an atom and the corresponding values of its variable.

Example

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

         port Port_t r(x), s(y)

  places START, SYNC, END

  initial             to START                 do { x=P; y=0; }
  internal from START to SYNC provided (x!=0) do { y=f(x); }
  on r     from START to SYNC                 do { y=x; }
  on s     from SYNC  to END
end
```

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 s 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**

```
atom_type_definition :=
  'atom type' atom_type_name '(' [ data_parameter (',' data_parameter)* ] ')'
     (['export'] 'data' data_type
        data_declaration_name ( ',' data_declaration_name )* )*
```


BIP2 Documentation, Release 2015.04 (RC7)

- modify the meta-model by editing the *ecorediag* (or the *ecore* directly).
- generate Java code (see below)
- implement constraints (if needed)

Meta-model organization

The meta-model is split in two parts:

- the *type model* is used to describe a BIP source code and nothing more: collections of types organized in packages.
- the *instance model* is used to describe a deployed system: instances of BIP types. This model points to the *type model*.

The instance model lives under the instance package. Everything else is related to the *type model*.

Generating Java code from the meta-model

Open the bip2.genmodel file in Eclipse, right-click on the single line named *Bip2* and select Generate Model code. This will generate code in the src/main/emf-generated directory.

Important: The directory src/main/emf-generated *is* versioned, please *review* the changes before commiting !

Constraints

A constraint is added on an element of the meta-model by adding an annotation:

- the *source* field for the annotation must be http://www.eclipse.org/emf/2002/Ecore
- then, an item with the key constraints contains a space separated list of constraint names.

When the java code is generated, EMF will create empty stubs that must be completed by the actual constraint code. In order to keep these changes even when the code generator is executed again, you *must* modify the comment before the constraint method. The convention adopted by most project is to add NOT (in capitals) after the @generated:

```
/**
 * Validates the constraintName constraint of '<em>Elt Name</em>'.
 * <!-- begin-user-doc -->
 * <!-- end-user-doc -->
 * @generated NOT
 */
```

Omitting this will end up in the loss of your changes during the next code generation execution.

The default code for error handling (*ie.* when a constraint is violated) must be changed to integrate well with the compiler error handling. By default, EMF produces the following code:

```
diagnostics.add(createDiagnostic(Diagnostic.ERROR,
                  DIAGNOSTIC_SOURCE, 0,
                  "_UI_GenericConstraint_diagnostic",
                  new Object[] {
                    "exportedDataListsSynchronized",
                    getObjectLabel(theElementWithAConstraint, context)
                  },
                  new Object[] {
```

When the module needs to do other actions, you need to override the `compile` target. This is the case for the metamodel, as java code is located in two different directories:

```
<target name="compile" depends="resolve" description="--> compile the project">
  <mkdir dir="${classes.dir}" />
  <javac srcdir="${src.dir}" destdir="${classes.dir}"
         classpathref="lib.path.id" debug="true">
    <src path="${src.dir}" />
    <src path="src/main/emf-generated" />
  </javac>
</target>
```

9.2.2 Eclipse

In order for all modules to be correctly imported in eclipse, you need to install the following *plug-ins*:

- Eclipse Modeling Framework (EMF): its part of eclipse and directly available in the *plug-ins* list.
- IvyIDE : you need to install this *plug-in* by following instructions available on the project webpage: http://ant.apache.org/ivy/ivyde/
- Acceleo : also available from the eclipse *plug-ins* list

Then, you simply need to use the *import existing project* of eclipse and point it to the directory containing the `Common`, `Frontend` and `Backend` directories. Eclipse should see all sub-project and import them.

Important: If you import projects from a fresh source tree, eclipse will fail at building the compiler because of missing java code in the *parser* project. Indeed, you need to build the ant target `gencode-for-eclipse`. See the description of the parser module for more details.

Important: It is **normal** that under the *projects* tab in the *build path* configuration windows, the list is empty. It should always be empty, as project dependencies are handled by the ivy pluggin. The only case where you need to add a dependency is when debugging a filter or back-end. This change must never be pushed to the code repository.

9.3 Front-end

9.3.1 `u.v.b.metamodel`

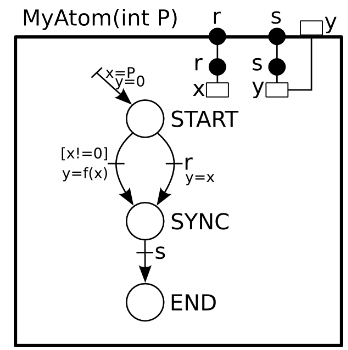
This module defines the BIP2 meta-model used by all parts of the compiler, as the meta-model is the intermediate representation of BIP models. It contains:

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

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.ecorediag` : 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:




```
  (['export'] 'port' port_type
     port_name '(' data_declaration_name (',' data_declaration_name )*) ')'
     ( ',' port_name '(' data_declaration_name (',' data_declaration_name )*) ')' )*
     ['as' port_name] )*
  'place' place_name (',' place_name)*
  'initial to' place_name (',' place_name)* ['do' actions]
  ( ('on' port_name | 'internal')
     'from' place_name (',' place_name)*
     'to' place_name (',' place_name)*
     ['provided' '(' transition_guard ')'] )*
     ['do' actions]
  atom_priority_declaration*
'end'
```

2.2.8 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`:

```
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 *Compounds*). 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 `exp` of type `port_t`, referencing a variable `tmp`, is declared in a connector type as follows:

```
export port port_t exp(tmp)
```

Defined interactions

Formally, an interaction of a connector type is a subset of its ports. A connector type explicitly define 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:

```
connector_port_expression :=
  ( sub_expression )+

sub_expression :=
  ( port_name | '(' connector_port_expression ')' ) [''']
```

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 ‘`p`‘. Interactions defined by an expression of the form `e'` are the ones defined by `e`. Interactions defined by an expression of the form `e1 e2...eN` are computed recursively from the interactions defined by sub-expressions `e1`, `e2`, ..., `eN`, as explained as follows. An interaction is defined by `e1 e2...eN` if both following rules apply:

- it can be written as a union of interactions defined by sub-expressions `e1`, `e2`, ..., `eN`
- it contains (at least) an interaction defined by a trigger sub-expression, or for each sub-expression `eI`, `I=0,...,N`, it contains an interaction defined by `eI`.

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

```
define (p q)' r s
```

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

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

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:

ivy.xml

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 `ujf.verimag.bip` module in `Common` shows the 2 types of dependencies:

```
<dependency name="joptsimple" rev="3.2">
  <artifact name="joptsimple" type="jar"
            url="file://${basedir}/externals/jopt-simple-3.2.jar" />
</dependency>
<dependency name="ujf.verimag.bip.error"
            rev="latest.integration"></dependency>
```

It defines 2 dependencies:

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

A single dependencies can have several artifacts, as is the case of the EMF in `ujf.verimag.bip.metamodel` module:

```
<dependency name="org.eclipse.emf" rev="2.7.0">
  <artifact name="org.eclipse.emf" type="jar"
            url="file://${basedir}/externals/org.eclipse.emf_2.6.0.v20110913-1156.jar"/>
  <artifact name="org.eclipse.emf.common" type="jar"
            url="file://${basedir}/externals/org.eclipse.emf.common_2.7.0.v20110912-0920.jar"/>
  <artifact name="org.eclipse.emf.ecore" type="jar"
            url="file://${basedir}/externals/org.eclipse.emf.ecore_2.7.0.v20110912-0920.jar"/>
  <artifact name="org.eclipse.emf.ecore.xmi" type="jar"
            url="file://${basedir}/externals/org.eclipse.emf.ecore.xmi_2.7.0.v20110520-1406.jar"/>
  <artifact name="org.eclipse.emf.mapping.ecore2xml" type="jar"
            url="file://${basedir}/externals/org.eclipse.emf.mapping.ecore2xml_2.7.0.v20110331-2022.
</dependency>
```

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

build.xml

This file is used by `ant` to schedule the build. This includes the actual compilation of source files (acceleo 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 `build.xml`:

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

9.1.1 Middle-end

The middle-end only contains the needed mechanics so that *filters* can be executed. One simple example is provided.

9.1.2 Common

- `ujf.verimag.bip` : contains elements shared by every parts of the compiler (mainly a java interface for *plug-in* mechanism)
- `ujf.verimag.bip.error` : the base of the error handling in all the compiler
- `ujf.verimag.bip.exception` : contains a single **unchecked** `CompilerErrorException` exception. This exception can be raised without having to explicitely declare it. It must be used only when a bug in the compiler has been detected. No recovery mechanism is present. The only handling done is to display a message to the user with the bare minium information to provide the developpers.

9.1.3 Front-end

- `ujf.verimag.bip.metamodel` : defines the BIP2 meta-model used in every bits of the compiler.
- `ujf.verimag.bip.parser` : defines the BIP2 grammar and the rules to build a *BIP-EMF* model from a BIP source
- `ujf.verimag.bip.instantiator` : builds an instance model from a type model and a *root* component definition
- `ujf.verimag.bip.userinterface.cli` : interacts with the user and instantiate all parts of the compiler and bind them together to form a coherent compiler
- `ujf.verimag.bip.frontend.tests` : JUnit tests for the previous parts

9.1.4 Back-end

- `ujf.verimag.bip.backend` : contains code shared by all *back-ends* (*eg.* some acceleo templates, back-end specific errors)
- `ujf.verimag.bip.backend.aseba` : ASEBA *back-end*
- `ujf.verimag.bip.backend.bip` : BIP *back-end*
- `ujf.verimag.bip.backend.cpp` : C++ *back-end*
- `ujf.verimag.bip.backend.example` : an example *back-end*, intended to be copied and used as a basis for new *back-end* development
- `ujf.verimag.bip.backend.tests` : JUnit for the previous parts
- `acceleo.standalone.compiler` : acceleo standalone compiler used to build the BIP compiler *outside* eclipse

9.2 Generalities

Before describing every internal parts of the compiler, we need to describe how the build system works and how to setup a correct development environment.

we define a rendez-vous interaction between two ports where a temporary value is stored in the `tmp` variable. To prevent division by zero, the interaction is disabled when the value of the `y` variable equals 0:

```
on p q provided (q.y != 0) up { tmp = p.x / q.y; }
```

The `down` instructions may update the values of the variables associated with the ports involved in an interaction. Port variables are assigned with values computed from connector variables and variables of the exported port. In the following example, the instructions swap the values of variables `x` and `y` of ports `p` and `q`:

```
on p q down { tmp = p.x; p.x = q.y; q.y = tmp; }
```

Notice that transfer functions `up` and `down` can be simultaneously defined for a connector interaction. `up` functions correspond to data moving upwards in the connector and component hierarchy, that is, from values of variables of the connected ports to the values of variables of the port exported by a connector. Once an interaction is chosen and executed, `down` functions correspond to the downward flow of data, that is, from variables of the exported port to variables of the connected ports.

Important: For a given interaction, the temporary values of connector variables when executing the `down` instructions are computed by the corresponding `up` instructions. However, these values are not accessible between different executions of the same interaction or between the execution of the transfer functions of different interactions. They are only stored between the execution of `up` and `down` instructions of the same interaction.

Example

```
connector type ConnectT(Port_t1 p, Port_t2 q, Port_t3 r)
  data int tmp
  export port Port_t exp(tmp)

  define p' q r

  on p     up { tmp = p.x; }                  down { p.x = tmp; }
  on p q   up { tmp = max(p.x, q.y); }        down { p.x = tmp; q.y = tmp; }
  on p r   up { tmp = max(p.x, r.z); }        down { p.x = tmp; r.z = tmp; }
  on p q r up { tmp = max(p.x, p.x, r.z); } down { p.x = tmp; q.y = tmp; r.z = tmp; }
end
```

In the previous BIP2 code excerpt we provided a complete definition of a connector type named `ConnectT` that connects three ports `p`, `q` and `r`. We have already seen in previous examples the enabled interactions and the computations performed by the transfer functions. A noticeable difference is that variable `tmp` is accessible to other connectors that interact with port `exp`. Hence, the value of `tmp` may differ from the computation performed by `up` since it may be altered by the transfer functions of connectors connected to `exp`. A simplified graphical representation of `ConnectT` is provided below.

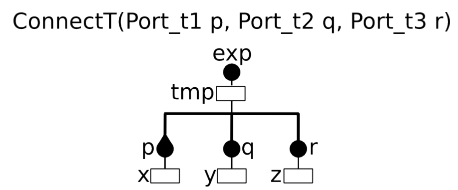


Figure 2.6: Connector type example

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`:

```
define p' q r
```

Based on the definition above, the permitted interactions are (‘`p`‘, ‘`p,q`‘, ‘`p,r`‘ and ‘`p,q,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 `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 `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 ‘`p`‘, ‘`p,q`‘, ‘`p,r`‘ and ‘`p,q,r`‘ are enabled. Moreover, there are 24 possible values for variable `tmp` of the exported port `exp`, corresponding to the application of `up` to all combinations of values for the 4 enabled interactions:

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

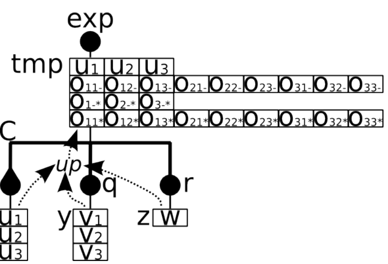


Figure 2.7: Enabled interaction of a connector and the corresponding values of variable `tmp`.

Syntax

- **accepts annotations**

```
connector_type_definition :=
  'connector type' connector_type_name '(' port_parameter (',' port_parameter )* ')'
    ('data' data_type data_declaration_name ( ',' data_declaration_name )* )*
    ['export port' port_type
       port_name '(' data_param_declaration (',' data_param_declaration)* ')' ]
    'define' connector_port_expression
    connector_interaction*
```


CHAPTER

NINE

DEVELOPER REFERENCE FOR COMPILER

9.1 Compiler design

Goals:

- users/devs usually write different code generator : adding a new code generator should be as easy as possible
- users/devs 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 : *any to BIP-EMF* transformation. Takes any source code in a given language and translate it to *BIP-EMF*
- middle-end :*BIP-EMF to BIP-EMF*. Apply operations on a *BIP-EMF* input (operations can be read and/or write).
- back-end : *BIP-EMF to any*. Generates source code in a given language from a *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, *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:

```
.
÷-- Middleend
|   ÷-- ujf.verimag.bip.middleend
|   `-- ujf.verimag.bip.middleend.example
÷-- Backend
|   ÷-- acceleo.standalone.compiler
|   ÷-- ujf.verimag.bip.backend
|   ÷-- ujf.verimag.bip.backend.aseba
|   ÷-- ujf.verimag.bip.backend.bip
|   ÷-- ujf.verimag.bip.backend.cpp
|   ÷-- ujf.verimag.bip.backend.example
|   `-- ujf.verimag.bip.backend.tests
÷-- Common
|   ÷-- ujf.verimag.bip
|   `-- ujf.verimag.bip.error
`-- Frontend
    ÷-- ujf.verimag.bip.frontend.tests
    ÷-- ujf.verimag.bip.instantiator
    ÷-- ujf.verimag.bip.metamodel
    ÷-- ujf.verimag.bip.parser
    `-- ujf.verimag.bip.userinterface.cli
```

We give here a very brief description of each module. Full description is given in the following sections.

```
    : (and_expression)
      (BWISE_XOR_OP exclusive_or_expression)?
    ;

and_expression
    : (equality_expression)
      (BWISE_AND_OP and_expression)?
    ;

equality_expression
    : relational_expression ((EQ_OP^|NE_OP^) relational_expression)?
    ;

relational_expression
    : additive_expression ((LT_OP^|GT_OP^|LE_OP^|GE_OP^) additive_expression)?
    ;

additive_expression
    : subtractive_expression
      (PLUS_OP additive_expression)*
    ;

subtractive_expression
    : multiplicative_expression
      (MINUS_OP subtractive_expression)*
    ;

multiplicative_expression
    : unary_expression ((DIV_OP^|MOD_OP^|MULT_OP^) unary_expression)*
    ;

unary_expression
    : (MINUS postfix_expression)
    | (bwise_unary_operator | logical_unary_operator)? postfix_expression
    ;

postfix_expression
    : primary_expression
    | function_call_expression
    ;

function_call_expression
    : fully_qualified_name LPAREN argument_expression_list? RPAREN
    ;

argument_expression_list
    : logical_or_expression (COMMA logical_or_expression)*
    ;
```



```
  'end'

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:A1.p1,A2.p2,...,AN.pN` where `C` is a connector and `A1.p1,A2.p2,...,AN.pN` 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:J` 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 Compound_t()
  component CompT1 K1()
```



```
annotation
    : AT ID (LPAREN annotation_param (COMMA annotation_param)* RPAREN)?
    ;

annotated_type_definition
    : annotation* type_definition
    ;

type_definition
     : atom_type_definition
     | compound_type_definition
     | port_type_definition
     | connector_type_definition
     ;

primary_expression
    : fully_qualified_name
    | INT
    | FLOAT
    | STRING
    | TRUE
    | FALSE
    | LPAREN! logical_or_expression RPAREN!
    ;

statement
    : assignment_expression
    | postfix_expression
    ;

if_then_else_expression
    : IF LPAREN logical_or_expression RPAREN
      THEN ((statement SEMICOL)|if_then_else_expression)+
      (ELSE ((statement SEMICOL)|if_then_else_expression)+)?
      FI
    ;

assignment_expression
    : postfix_expression ASSIGN_OP^ logical_or_expression
    ;

logical_or_expression
    : (logical_and_expression)
      (OR_OP logical_or_expression)?
    ;

logical_and_expression
    : (inclusive_or_expression)
      (AND_OP logical_and_expression)?
    ;

inclusive_or_expression
    : (exclusive_or_expression)
      (BWISE_OR_OP inclusive_or_expression)?
    ;

exclusive_or_expression
```

```
    : LPAREN connector_port_expression RPAREN QUOTE?
    ;

connector_port_expression
    : (port_primary_expression | port_nested_expression)+
    ;

port_type_param
    : fully_qualified_name simple_name
    ;

fragment_port_declaration
    : simple_name LPAREN (simple_name (COMMA simple_name)*)? RPAREN
    ;

multi_port_declaration_with_modifiers
    : annotation*
      (EXPORT)? multi_port_declaration (AS simple_name)?
    ;

multi_port_declaration
    : PORT fully_qualified_name fragment_port_declaration
      (COMMA fragment_port_declaration)*
    ;

single_port_declaration
    : PORT fully_qualified_name fragment_port_declaration
    ;

connector_provided_expression
    : LPAREN logical_or_expression RPAREN
    ;

connector_action
    : ((statement SEMICOL!)| if_then_else_expression)+
    ;

connector_interaction
    : annotation*
      ON simple_name+
      (PROVIDED connector_provided_expression)?
      (UP_ACTION LBRACE connector_action? RBRACE)?
      (DOWN_ACTION LBRACE connector_action? RBRACE)?
    ;

connector_type_definition
    : CONNECTOR TYPE simple_name
      LPAREN (port_type_param (COMMA port_type_param)*)  RPAREN
      multi_data_declaration*
      (EXPORT single_port_declaration)?
      DEFINE connector_port_expression
      connector_interaction*
      END
    ;

annotation_param
    : ID (ASSIGN_OP (ID|TRUE|FALSE|STRING))?
    ;
```



```
  component CompT2 K2()
  component CompT3 K3()

  connector BRDXP C(K1.p, K2.q)
  connector RDVXP D(C.xp, K3.t)
  connector RDV   E(K2.q, K3.s)

  export port C.xp as u
  export port F.xp as v
  export port K3.t as w

  export data K3.x as x
end
```

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`).

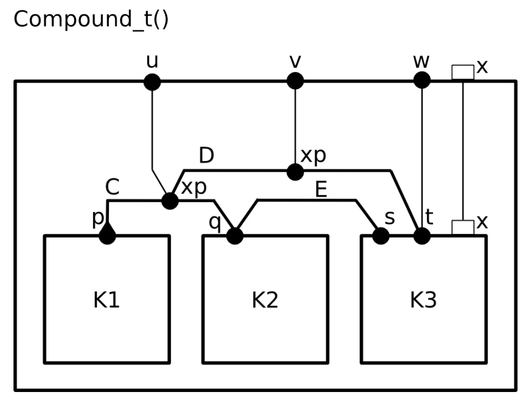


Figure 2.8: Example of a compound type.

Syntax

- **accepts annotations**

```
compound_type :=
  'compound type' compound_type_name '(' [data_parameter (',' data_parameter)*] ')'
    component_declaration+
    connector_declaration*
    compound_priority_declaration*
    inner_port_export*
    inner_data_export*
  'end'

inner_port_export :=
  'export port' port_reference (',' port_reference)* 'as' exported_name

inner_data_export :=
  'export data' data_reference 'as' exported_name

compound_priority_declaration :=
  'priority' priority_name
      ('*:*' | compound_interaction) '<' ('*:*' | compound_interaction)
      [ 'provided' compound_priority_guard ]

compound_interaction :=
  connector_name ':' ('*' | (port_reference (',' port_reference)*))
```

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.


```
    ;


compound_type_definition
    : COMPOUND TYPE simple_name
      LPAREN (comp_type_data_params)? RPAREN
      multi_component_declaration+
      multi_connector_declaration*
      compound_priority_declaration*
      export_inner_port*
      export_inner_data*
      END
    ;

native_data_type_name
    : CT_INT
    | CT_BOOL
    | CT_FLOAT
    | CT_STRING
    ;

data_type_name
    : fully_qualified_name
    | native_data_type_name
    ;

native_data_type_param
    : native_data_type_name simple_name
    ;

any_data_type_param
    : data_type_name simple_name
    ;

multi_data_declaration_with_modifiers
    : annotation*
      EXPORT? multi_data_declaration
    ;

multi_data_declaration
    : DATA data_type_name simple_name (COMMA simple_name)*
    ;


port_type_data_params
    : any_data_type_param (COMMA any_data_type_param)*
    ;

port_type_definition
    : PORT TYPE simple_name
      LPAREN (port_type_data_params)? RPAREN
    ;

port_primary_expression
    : simple_name QUOTE?
    ;

port_nested_expression
```

```
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
```


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.

```
      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)?
    ;
```

```
    : BWISE_AND_OP | BWISE_OR_OP | BWISE_XOR_OP
    ;

bwise_unary_operator
    : BWISE_NOT_OP
    ;

logical_binary_operator
    : AND_OP | OR_OP
    ;

logical_unary_operator
    : NOT_OP
    ;

fully_qualified_name
    : ID (DOT ID)*
    ;

simple_name
    : ID
    ;

bip_package
    : annotation*
      PACKAGE fully_qualified_name
      (USE fully_qualified_name)*
      annotated_const_data_declaration*
      annotated_extern_data_type*
      annotated_extern_prototype*
      annotated_type_definition*
      END
    ;

annotated_extern_prototype
    : annotated_extern_function_prototype
    | annotated_extern_binary_operator_prototype
    | annotated_extern_unary_operator_prototype
    ;

annotated_extern_data_type
    : annotation* EXTERN DATA TYPE simple_name
      (REFINE data_type_name (COMMA data_type_name)*)?
      (AS STRING)?
    ;

annotated_extern_function_prototype
    : annotation* EXTERN FUNCTION
      data_type_name? simple_name LPAREN data_types_params? RPAREN
    ;

annotated_extern_binary_operator_prototype
    : annotation* EXTERN OPERATOR data_type_name binary_operator
      LPAREN data_type_name COMMA fully_qualified_name RPAREN
    ;

annotated_extern_unary_operator_prototype
    : annotation* EXTERN OPERATOR data_type_name unary_operator
```


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 *filter* later on.
- 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.

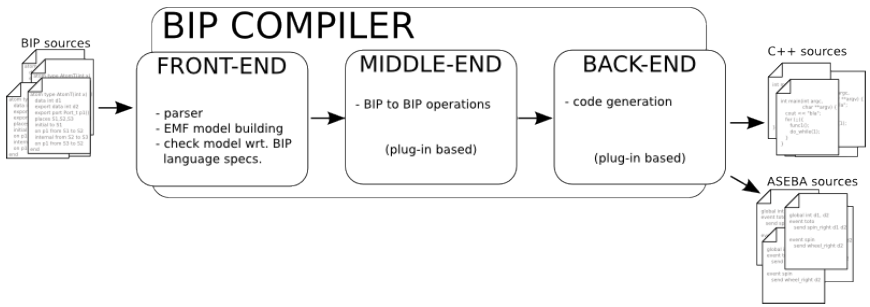


Figure 3.1: Overview of Compiler design

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 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 (*ie.* the front-end) is then able to build both a type model (*ie.* 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:

  ```
  public class MyClass { ... }
  ```

- a component (instance of a component type) = an object (instance of a class):

  ```
  new MyClass();
  ```

- a component declaration = an object declaration:

  ```
  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 (*eg.* 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 (*eg.* 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).


```
    |   '\\' ('0'..'7') ('0'..'7')
    |   '\\' ('0'..'7')
    ;

fragment
UNICODE_ESC
    :   '\\' 'u' HEX_DIGIT HEX_DIGIT HEX_DIGIT HEX_DIGIT
    ;

LT_OP   :   '<';
GT_OP   :   '>';
LE_OP   :   '<=';
GE_OP   :   '>=';
EQ_OP   :   '==';
NE_OP   :   '!=';
AND_OP  :   '&&';
OR_OP   :   '||';
NOT_OP  :   '!';

PLUS_OP :   '+';
MINUS_OP:   '-';
MULT_OP :   '*';
DIV_OP  :   '/';
MOD_OP  :   '%';

BWISE_AND_OP : '&';
BWISE_OR_OP : '|';
BWISE_XOR_OP : '^';
BWISE_NOT_OP : '~';

ASSIGN_OP : '=';

binary_operator
    : comparison_operator
    | arithmetic_binary_operator
    | bwise_binary_operator
    | logical_binary_operator
    ;

unary_operator
    : arithmetic_unary_operator
    | bwise_unary_operator
    | logical_unary_operator
    ;

comparison_operator
    : EQ_OP | NE_OP | GT_OP | GE_OP | LT_OP | LE_OP
    ;

arithmetic_binary_operator
    : PLUS_OP | MINUS_OP | MULT_OP | DIV_OP | MOD_OP
    ;

arithmetic_unary_operator
    : PLUS_OP | MINUS_OP
    ;

bwise_binary_operator
```

```
RBRACE  :   '}';
COMMA   :   ',';
QUOTE   :   '\'';
DOT     :   '.';
SEMICOL :   ';';
COLON   :   ':';
AT      :   '@';

IF : 'if';
THEN : 'then';
ELSE : 'else';
FI : 'fi';


ID  :   ('a'..'z'|'A'..'Z'|'_') ('a'..'z'|'A'..'Z'|'0'..'9'|'_')*
    ;

INT :   '0'..'9'+
    ;

FLOAT
    :   ('0'..'9')+ DOT ('0'..'9')* EXPONENT?
    |   DOT ('0'..'9')+ EXPONENT?
    |   ('0'..'9')+ EXPONENT
    ;

COMMENT
    :   '//' ~('\n'|'\r')* '\r'? '\n' {$channel=HIDDEN;}
    |   '/*' ( options {greedy=false;} : . )* '*/' {$channel=HIDDEN;}
    ;

WS  :   ( ' '
        | '\t'
        | '\r'
        | '\n'
        ) {$channel=HIDDEN;}
    ;

STRING
    :  '"' ( ESC_SEQ | ~('\\'|'"') )* '"'
    ;

fragment
EXPONENT : ('e'|'E') ('+'|'-')? ('0'..'9')+ ;

fragment
HEX_DIGIT : ('0'..'9'|'a'..'f'|'A'..'F') ;

fragment
ESC_SEQ
    :   '\\' ('b'|'t'|'n'|'f'|'r'|'\"'|'\''|'\\')
    |   UNICODE_ESC
    |   OCTAL_ESC
    ;

fragment
OCTAL_ESC
    :   '\\' ('0'..'3') ('0'..'7') ('0'..'7')
```


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* of 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* of 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* of 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).

CHAPTER

EIGHT

BIP 2 GRAMMAR

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

```
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';
PACKAGE :   'package';
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  :   '{';
```

```
      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 transation 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          }
   |15          on p from S to END do {
B+>|16            my_print("Hello World", somedata);
   |17          }
   |18        end
   |19
   |20          compound type HelloCompound()
   |21          component HelloAtom c1()
```

You can of course set breakpoint in your external code:

```
(gdb) b HelloPackage.cpp:8
Breakpoint 2 at 0x80665a8: file /path/to/debug_bip_level/ext-cpp/HelloPackage.cpp, line 8.
```


CHAPTER

FOUR

INSTALLING & USING THE BIP COMPILER

4.1 Requirements

BIP compiler is currently only tested on GNU/Linux systems. It is know to work correctly on Mac OSX, 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 (*eg.* Ubuntu), you can install this dependency with:

```
$ apt-get install openjdk-6-jre
```

Warning: Theses instructions covers the installation of the compiler. The common usage involves the generation of C++ code and need 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 (eg. `i686`). Then simply follow the following steps:

- create a directory where everything will be installed:

  ```
  $ mkdir bip2
  ```

- extract the archive:

```
$ cd bip2 ; tar zxvf /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 environement (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 zxvf /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:


```
  return o;
}
```

Compile the code *without* the `--gencpp-no-serial`:

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

We can check that our serialization code is correctly use by reading the execution trace:

```
[BIP ENGINE]: initialize components...
Data for: 1 = 2,8
Data for: 2 = 4,16
Data for: 3 = 6,24
[BIP ENGINE]: state #0: 1 interaction:
[BIP ENGINE]:   [0] ROOT.connect: ROOT.c1.p({d}=[2, 8];) ROOT.c2.p({d}=[4, 16];) ROOT.c3.p({d}=[6, 2
[BIP ENGINE]: -> choose [0] ROOT.connect: ROOT.c1.p({d}=[2, 8];) ROOT.c2.p({d}=[4, 16];) ROOT.c3.p({
Data for: 1 = 6,24
Data for: 2 = 6,24
Data for: 3 = 6,24
[BIP ENGINE]: state #1: deadlock!
```

Important: In this example, we used a *regular* C *struct* type, but you can of course use C++ classes (which are basically the same as *structs*).

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(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
    port HelloPort_t p()
    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 {
```

```
Data for: 3 = 6,24
[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(-)
Data for: 1 = 6,24
Data for: 2 = 6,24
Data for: 3 = 6,24
[BIP ENGINE]: state #1: deadlock!
```

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 << "[" << value.x << ", "  << value.y << "]";
```



```
.
÷-- bin
|   `-- bipc.sh
`-- lib
    ÷-- acceleo.standalone.compiler_1.0-20120102155443.jar
    ÷-- apache.tool.ant_1.8.0.jar
    ÷-- backends
    |   ÷-- ujf.verimag.bip.backend.aseba_1.0-20120102155513.jar
    |   ÷-- ujf.verimag.bip.backend.bip_1.0-20120102155537.jar
    |   `-- ujf.verimag.bip.backend.cpp_1.0-20120102155558.jar
    ÷-- com.google.collect_1.0.0.v201105210816.jar
    ÷-- filters
    ÷-- joptsimple_3.2.jar
    ÷-- lpg.runtime.java_2.0.17.v201004271640.jar
    ...
```

- the `bin` directory contains the compiler's executables. Usually, there is only the `bipc.sh` script used to run the compiler.
- the `lib` directory contains all java dependencies for the compiler. The sub-directory `backends` contains the back-end installed with the compiler. The `filters` contains the filter composing the middle-end. All files outside this sub-directory are libraries used by the compiler (EMF, eclipse runtime, command line parsing, ...)

4.3 Front-end checks for BIP model correctness

The compiler always checks if a given input is valid with respect to the language (*eg.* 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 the 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 message 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 message 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)
```

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 (*ie.* 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 (*eg.* `-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:

```
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:
    109:  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:

```
@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


```
  connector type ThreeRendezVous(HelloPort_t p1, HelloPort_t p2, HelloPort_t p3)
  define p1 p2 p3
  on p1 p2 p3 down { p1.d = p3.d; p2.d = p3.d; }
  end

  compound type HelloCompound()
    component HelloAtom c1(1), c2(2), c3(3)
    connector ThreeRendezVous connect(c1.p, c2.p, c3.p)
  end
end
```

`HelloPackage.hpp`:

```
#ifndef HP_HPP
#define HP_HPP

typedef struct {
  int x,y;
} 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 <iostream>
#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;
}
```

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
```

```
$ 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 executing, we can see that the transition for the *positive* transition is fired:

```
[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__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).

The source code files are given below.

`HelloPackage.bip`:

```
@cpp(src="ext-cpp/HelloPackage.cpp",include="HelloPackage.hpp")
package HelloPackage
  extern data type my_custom_type
  extern function init_data(int, my_custom_type)
  extern function print_data(int, my_custom_type)

  port type HelloPort_t(my_custom_type d)

  atom type HelloAtom(int id)
    data my_custom_type d
    export port HelloPort_t p(d)
    place START,END
    initial to START do {init_data(id, d); print_data(id, d);}
    on p from START to END do {print_data(id, d);}
  end
```


- compoundprioritycycle
- uselessdown
- nointeraction
- missinginteraction
- modifiedvariabletransition
- modifiedvariableinteraction

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 directoy, you must use the ''-I" parameter to add the directories that contain them. For example:

- you are developping 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
        `-- Foo.bip
```

```
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 petrinet 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"
```

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 *1 filter by line* convention, as the line feed is ignored:

```
filter1_name foo=bar foo2=bar2 !
filter2_name bla=bar !
filter3_name some_very_complex_arg=something_very_very_long
```

And simply give this file to the compiler:

```
bipc.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                       Add a path to the include search path
                                      (used when calling the C++ compiler)
--gencpp-cc-extra-src               Add an extra source file to the
                                      compilation list
--gencpp-follow-used-packages       Also generate code for used packages.
--gencpp-ld-L                       Add a path to the libraries search
                                      path (used when calling the linker)
--gencpp-ld-extra-obj               Add an extra object file to be linked
                                      with the other parts
--gencpp-ld-l                       Link with this library (use several
                                      times to add many libraries)
--gencpp-no-serial                  Disable the generation of
                                      serialization code
--gencpp-output-dir                 Output directory for CPP backend
--gencpp-optim                      Set the optimization level (defaults
                                    to none = 0). Each level includes a
                                    set of optimization.
--gencpp-set-optim-param            Set an optimisation parameter:
                                    optimname:key:value
--gencpp-disable-optim              Disable a specific optimization (can
                                    be used several times)
--gencpp-enable-optim               Enable a specific optimization (can be
                                    used several times)
--gencpp-enable-bip-debug           Generates extra code to enable GDB to
                                    debug at the BIP level
```

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:

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(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()
    place START, S, END
    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

  compound type HelloCompound()
    component HelloAtom c1()
  end
end
```

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 corresponding `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;
}

bool const_my_guard(int &adata){
  return adata > 0;
}
```

The compilation is still the same:

```
  extern function my_print(string, int)

  port type HelloPort_t()

  atom type HelloAtom()
    data int somedata
    port HelloPort_t p()
    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 { my_print("Hello World", somedata);}
  end

  compound type HelloCompound()
    component HelloAtom c1()
  end
end
```

The new `HelloPackage.hpp`:

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

And the corresponding `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;
}
```

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 that 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!
```



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

4.5.2 BIP back-end

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 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 are being 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.

```
    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 c1()
  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`:

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

`HelloPackage.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:

```
$ 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(src="ext-cpp/HelloPackage.cpp",include="HelloPackage.hpp")
package HelloPackage
  extern function my_modify(int)
```

- the implementation (*ie.* `HelloPackage.cpp`) corresponding to the previous interface.

Any directory layout can be used. We propose the following as example:

```
.
÷-- ext-cpp
|   ÷-- HelloPackage.cpp
|   `-- HelloPackage.hpp
`-- HelloPackage.bip
```

With the following content for `HelloPackage.hpp`:

```
void my_print(const char *message);
```

And for `HelloPackage.cpp`:

```
#include <iostream>

void my_print(const char *message){
  std::cout << "Someone says: " << message;
}
```

Then, compile it using the following commands:

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

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

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

  port type HelloPort_t()

  atom type HelloAtom()
```


CHAPTER

FIVE

MORE ABOUT C++ CODE GENERATOR

5.1 Presentation & prerequisites

5.1.1 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 directory `/home/a_user/output` should contain several files & directories:

```
.
÷-- CMakeLists.txt
÷-- Deploy
÷   ÷-- Deploy.cpp
|   ÷-- Deploy.hpp
|   `-- DeployTypes.hpp
÷-- SamplePackage
|   ÷-- CMakeLists.txt
|   ÷-- include
|   |   `-- SamplePackage
|   |       ÷-- 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` : adds a directory to the compiler search paths for *include* files (*ie.* this is the `-I` used by most C++ compilers)

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


```
  end

  compound type HelloCompound()
    component HelloAtom c1()
  end
end
```

The annotation instructs the code generator to include the `stdio.h` C standard library in the generated code for this package. This allows the use of `printf()`.

The compilation stays the same:

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

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]:   [0] ROOT.c1._iport_decl__p
[BIP ENGINE]:  -> choose [0] ROOT.c1._iport_decl__p
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")
package HelloPackage
  extern function my_print(string)

  port type HelloPort_t()

  atom type HelloAtom()
    port HelloPort_t p()
    place START,END
    initial to START
    on p from START to END do { my_print("Hello World!\n"); }
  end

  compound type HelloCompound()
    component HelloAtom c1()
  end
end
```

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.

```
      do { free = 0; }

    on utilize from WAIT to FREE
      do { free = 1; }

    on notfree from FREE to WAIT
      do { free = 0; }
  end
```

The corresponding execution involves both components `A` and `B`. `A` can only be executed when component `B` is in place `FREE`.

```
...
[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]: -> choose [0] ROOT.B_utilize_R: ROOT.B.utilize() ROOT.R.utilize()
[BIP ENGINE]: state #1: 1 interaction and 1 internal port:
[BIP ENGINE]:   [0] ROOT.A_utilize_R: ROOT.A.utilize() ROOT.R.utilize()
[BIP ENGINE]:   [1] ROOT.B._iport_decl__notfree
[BIP ENGINE]:  -> choose [0] ROOT.B._iport_decl__notfree
[BIP ENGINE]: state #2: 1 interaction:
[BIP ENGINE]:   [0] ROOT.B_utilize_R: ROOT.B.utilize() ROOT.R.utilize()
[BIP ENGINE]: -> choose [0] ROOT.B_utilize_R: ROOT.B.utilize() ROOT.R.utilize()
[BIP ENGINE]: state #3: 1 interaction and 1 internal port:
[BIP ENGINE]:   [0] ROOT.A_utilize_R: ROOT.A.utilize() ROOT.R.utilize()
[BIP ENGINE]:   [1] ROOT.B._iport_decl__notfree
[BIP ENGINE]:  -> choose [0] ROOT.B._iport_decl__notfree
[BIP ENGINE]: state #4: 1 interaction:
[BIP ENGINE]:   [0] ROOT.B_utilize_R: ROOT.B.utilize() ROOT.R.utilize()
[BIP ENGINE]: -> choose [0] ROOT.B_utilize_R: ROOT.B.utilize() ROOT.R.utilize()
[BIP ENGINE]: state #5: 1 interaction and 1 internal port:
[BIP ENGINE]:   [0] ROOT.A_utilize_R: ROOT.A.utilize() ROOT.R.utilize()
[BIP ENGINE]:   [1] ROOT.B._iport_decl__notfree
[BIP ENGINE]: -> choose [0] ROOT.A_utilize_R: ROOT.A.utilize() ROOT.R.utilize()
...
```

7.6 Using the C++ back-end

7.6.1 Hello World using a preinstalled library

The initial *Hello World* example does not display anything on its own. In this example, we add such simple display by using the common `printf()` from standard C library.

Change the initial example to match the following:

```
@cpp(include="stdio.h")
package HelloPackage
  extern function printf(string)

  port type HelloPort_t()

  atom type HelloAtom()
    port HelloPort_t p()
    place START,END
    initial to START
    on p from START to END do { printf("Hello World!\n"); }
```


files will be merged with the compiled code inside the library (*ie.* `.a` file) for the package. You can also add *include* directives that will be added to type/package generated files.

You need to use annotations in the BIP source file (see *Debugging*).

At the global level

You can inject source or object code at the global level or force the linking with an external library. Source code injected at this level will be compiled after all packages have been compiled. Object code or library are simply linked with all the other compiled code.

To achieve this integration, you can use the following parameters:

- `--gencpp-cc-extra-src` : adds a source file in the compilation process.
- `--gencpp-ld-L` : adds a directory to the linker search paths for libraries (*ie.* this is the `-L` used by most linkers)
- `--gencpp-ld-l` : adds a library to the link list (*ie.* this is the `-l` used by most linkers)
- `--gencpp-ld-extra-obj` : adds an object file to the link list

5.3.2 Data handling

It is possible to use data when calling external C++ code. There are two important facts to keep in mind:

- It is important to understand in which context the call is made as the function being called depends on that.
- A function call is NEVER type-checked by the BIP compiler. It means that you can easily write WRONG code. Hopefully, your C++ compiler will catch bad cases (but don't rely on that). A function call can take data parameters and can return a single data value.

For context where the callee can change the data (*ie.* connector `down{}` and petrinet transition `do{}`):

- function call `f()` in BIP is mapped to a C++ function call `f()`.
- the BIP assignment `x = f()` is mapped to the equivalent C++ `corresponding_internal_data_var = f()`. Type compatibility checked by C++ compiler.
- function call with data argument `f(a,b,c)` with `a`, `b` and `c` local BIP data declared in the caller's scope (atom, connector) is mapped in C++ to `f(internal_data_a, internal_data_b, internal_data_c)`. Expected prototype for f: `f(T1 &a, T2 &b, T3 &c)`.
- the BIP assignment `x = f(a,b,c)` is mapped to C++ `internal_data_x = f(internal_data_a, internal_data_b, internal_data_c)`, with expected prototype: `T1 f(T2 &a, T3 &b, T4 &c)`. Beware that the return type is not a reference nor a pointer. If you need to avoid useless copy, you can have the output variable be a parameter and modify it from within the function body (*ie. by-reference* parameter).

For context where the data used must *not* be modified (*ie.* const context: `up{}` and `provided()`), all function call are prefixed by `const_`:

- function call `f()` is mapped in C++ to `const_f()`
- `x = f()` is mapped to `corresponding_internal_data_var = const_f()`. Type compatibility checked by C++ compiler.
- `f(a,b,c)` with `a`, `b` and `c` local data declared in the caller's scope (atom, connector) , mapped to `const_f(internal_data_a, internal_data_b, internal_data_c)`. Expected prototype for f: `f(const T1 &a, const T2 &b, const T3 &c)`. The const are only expected. If `const_f()`

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 `internal_data_x = const_f(internal_data_a, internal_data_b, internal_data_c)`, 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 between *const* and *non-const* function 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:

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

The function must look like:

```
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:

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

`f` should have the following prototype:


```
    place WAIT, FREE

    initial to WAIT
      do { free = 0; }

    on utilize from WAIT to FREE
      do { free = 1; }

    internal from FREE to WAIT
      do { free = 0; }
  end

  connector type RDV(Port p, Port q)
    define p q
  end

  compound type Model()
    component Resource R()
    component UserOfRessource A(), B()

    connector RDV A_utilize_R(A.utilize, R.utilize)
    connector RDV B_utilize_R(B.utilize, R.utilize)

    priority scheduler A_utilize_R:* < *:* provided (B.free == 0)
  end
end
```

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]: -> choose [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.B_utilize_R: ROOT.B.utilize() ROOT.R.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.B_utilize_R: ROOT.B.utilize() ROOT.R.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.B_utilize_R: ROOT.B.utilize() ROOT.R.utilize()
...
```

The problem can be fixed by using a transition labelled by an internal port instead of an internal transition. A correct version of `UserOfRessource` is provided below.

```
atom type UserOfRessource()
   export data int free
   port Port notfree()
   export port Port utilize()

   place WAIT, FREE

   initial to WAIT
```

```
...
[BIP ENGINE]: initialize components...
[BIP ENGINE]: state #0: 1 interaction:
[BIP ENGINE]:   [1] ROOT.brdDEF: ROOT.D.p() ROOT.E.p() ROOT.F.p()
[BIP ENGINE]: -> choose [0] ROOT.brdDEF: ROOT.D.p() ROOT.E.p() ROOT.F.p()
[BIP ENGINE]: state #1: 1 interaction:
[BIP ENGINE]:   [0] ROOT.brdABC: ROOT.A.p() ROOT.B.p()
[BIP ENGINE]: -> choose [0] ROOT.brdABC: ROOT.A.p() ROOT.B.p()
[BIP ENGINE]: state #2: deadlock!
```

Notice that after initialization, only interaction ‘`D.p,E.p,F.p`‘ is listed by the engine, since it can only executes maximal interactions. Replacing the priority rule `scheduler` by `brdABC:A.p,B.p,C.p < brdDEF: D.p` leads to the same execution sequence. This is due to the fact that priorities are computed as the transitive closure of the union of maximal progress and the priority rules provided by `priority` statements. Even if interactions ‘`A.p,B.p,C.p`‘ is not enabled by `brdABC`, and interaction `D.p` is not enabled by `brdDEF`, priority rule `brdABC:A.p,B.p < brdDEF: D.p,E.p,F.p` is deduced from maximal progress that enforces `brdABC:A.p,B.p < brdABC:A.p,B.C.p` and `brdDEF: D.p < brdDEF:D.p,E.p,F.p`, and from `brdABC:A.p,B.p,C.p < brdDEF: D.p`.

Notice also that priority rules must only involve interactions that are defined by the connectors (*i.e.* by the expression provided with the statement `define`). As a result, if the priority rule `scheduler` is replaced by `brdABC:A.p,B.p,C.p < brdDEF: E.p`, an error is reported when compiling the model:

```
[SEVERE] In /home/to/example/priorities_in_compound.bip:
Interaction not allowed as not defined by connector type :
    26:
    27:     priority scheduler brdABC:A.p,B.p,C.p < brdDEF:E.p
----------------------------------------------------^
    28:   end
    29: end
```

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 set 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`.

```
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 UserOfRessource()
    export data int free
    export port Port utilize()
```



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

If you use

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

The code will run, but all modifications of `x` within the `f` function will be lost when the function returns. It will also have an overhead as data will be copied at invocation.

If the function takes a data from the type parameter, like the following:

```
atom type AT(int x1)
  data in x
  ...
  on p from S to T do {f(x, x1, 1+4);}
  ...
end
```

`f` should have the following prototype:

```
void f(int &a, int b, int c);
```

5.4 Parameters

- `--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 optimization 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:

```
$ bipc.sh ... --gencpp-enable-optim poolci \
  --gencpp-set-optim-param 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 `poolciv` 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.
- incoherences/unexpected debuger behavior can appear, as the mapping is not necessarily bijective (*eg.* 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 scripts:

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

5.7 Annotations

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

- **scope** : package definition, any type definition


```
    on p provided (false)
  end

  compound type Model()
    component MyAtom A(1), B(1), C(0)
    component MyAtom D(1), E(1), F(1)

    connector Broadcast brdABC(A.p, B.p, C.p)
    connector Broadcast brdDEF(D.p, E.p, F.p)
  end
end
```

In the above example, we synchronize components `A`, `B`, `C`, `D` `E`, `F` using two connectors `brdABC` and `brdDEF` of type `Broadcast`. Since ports `p` of `A` is considered as a trigger in connector `brdABC`, `brdABC` defines (statically) interactions '`A.p`', '`A.p,B.p`', '`A.p,C.p`' and '`A.p,B.p,C.p`'. Similarly, `brdDEF` defines '`D.p`', '`D.p,E.p`', '`D.p,F.p`' and '`D.p,E.p,F.p`'. Due to the guard `false` in `Broadcast`, interactions '`A.p`' and '`D.p`' are disabled. Moreover, due to the guard `(enabled == 1)` in `MyAtom` and the parameter `0` of `C`, interactions `A.p,C.p`' and '`A.p,B.p,C.p`' are also disabled. As a result, interactions enabled after initialization are: '`A.p,B.p`' in `brdABC` and '`D.p,E.p`', '`D.p,F.p`' and '`D.p,E.p,F.p`' in `brdDEF`.

In BIP2, maximal progress is considered as default priorities. Given a connector, maximal progress gives higher priority to larger interactions. In the above example, interactions '`D.p,E.p`' and '`D.p,F.p`' of connector `brdDEF` are not maximal since a larger interaction—'`D.p,E.p,F.p`'—is enabled in the same connector. As a result, execution sequences of instances of `Model` corresponds to the execution of '`A.p,B.p`' and '`D.p,E.p,F.p`' in an arbitrary order, that is, either the following execution sequence if '`A.p,B.p`' is executed first:

```
...
[BIP ENGINE]: initialize components...
[BIP ENGINE]: state #0: 2 interactions:
[BIP ENGINE]:   [0] ROOT.brdABC: ROOT.A.p() ROOT.B.p()
[BIP ENGINE]:   [1] ROOT.brdDEF: ROOT.D.p() ROOT.E.p() ROOT.F.p()
[BIP ENGINE]: -> choose [0] ROOT.brdABC: ROOT.A.p() ROOT.B.p()
[BIP ENGINE]: state #1: 1 interaction:
[BIP ENGINE]:   [0] ROOT.brdDEF: ROOT.D.p() ROOT.E.p() ROOT.F.p()
[BIP ENGINE]: -> choose [0] ROOT.brdDEF: ROOT.D.p() ROOT.E.p() ROOT.F.p()
[BIP ENGINE]: state #2: deadlock!
```

or the following execution sequence if '`D.p,E.p,F.p`' is executed first:

```
...
[BIP ENGINE]: initialize components...
[BIP ENGINE]: state #0: 2 interactions:
[BIP ENGINE]:   [0] ROOT.brdABC: ROOT.A.p() ROOT.B.p()
[BIP ENGINE]:   [1] ROOT.brdDEF: ROOT.D.p() ROOT.E.p() ROOT.F.p()
[BIP ENGINE]: -> choose [1] ROOT.brdDEF: ROOT.D.p() ROOT.E.p() ROOT.F.p()
[BIP ENGINE]: state #1: 1 interaction:
[BIP ENGINE]:   [0] ROOT.brdABC: ROOT.A.p() ROOT.B.p()
[BIP ENGINE]: -> choose [0] ROOT.brdABC: ROOT.A.p() ROOT.B.p()
[BIP ENGINE]: state #2: deadlock!
```

Using priorities to enforce an order of execution

We can modify the previous example to enfore 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`:

```
priority scheduler brdABC:A.p,B.p < brdDEF:D.p,E.p,F.p
```

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

This cycle can only occur if both guards `((i%2) == 0)` and `((i%2) == 1)` evaluates to true for the same state, which can never happen (otherwise an error will be reported at run-time). The execution of the model corresponds to the expected behavior, that is, the alternation of the execution of `p` and `q`. Notice that in this case, the model defines also a single execution sequence, as follows:

```
...
[BIP ENGINE]: initialize components...
[BIP ENGINE]: state #0: 1 internal port:
[BIP ENGINE]:   [0] ROOT.a._iport_decl__p
[BIP ENGINE]:  -> choose [0] ROOT.a._iport_decl__p
[BIP ENGINE]: state #1: 1 internal port:
[BIP ENGINE]:   [0] ROOT.a._iport_decl__q
[BIP ENGINE]:  -> choose [0] ROOT.a._iport_decl__q
[BIP ENGINE]: state #2: 1 internal port:
[BIP ENGINE]:   [0] ROOT.a._iport_decl__p
[BIP ENGINE]:  -> choose [0] ROOT.a._iport_decl__p
[BIP ENGINE]: state #3: 1 internal port:
[BIP ENGINE]:   [0] ROOT.a._iport_decl__q
[BIP ENGINE]:  -> choose [0] ROOT.a._iport_decl__q
[BIP ENGINE]: state #4: 1 internal port:
[BIP ENGINE]:   [0] ROOT.a._iport_decl__p
[BIP ENGINE]:  -> choose [0] ROOT.a._iport_decl__p
[BIP ENGINE]: state #5: 1 internal port:
[BIP ENGINE]:   [0] ROOT.a._iport_decl__q
[BIP ENGINE]:  -> choose [0] ROOT.a._iport_decl__q
[BIP ENGINE]: state #6: 1 internal port:
[BIP ENGINE]:   [0] ROOT.a._iport_decl__p
[BIP ENGINE]:  -> choose [0] ROOT.a._iport_decl__p
...
```

If guards of priorities `myPrioEven` are `myPrioOdd` are enabled at the same state of the model an error is reported when executing the model, *e.g.* if both guards are `((i%2) == 0)` the execution is as follows:

```
...
[BIP ENGINE]: initialize components...
[BIP ENGINE]: ERROR: cycle in priorities! (p < q < p)
```

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.

```
package priorities_in_compound
  port type Port()

  atom type MyAtom(int enabled)
    export port Port p()

    place SYNC, END

    initial to SYNC

    on p from SYNC to END
      provided (enabled == 1)
  end

  connector type Broadcast(Port p, Port q, Port r)
    define p' q r
```


- **argument** : comma separated list of file names
- **role** [the files specified as argument will be inserted in the file list] used during the compilation process along with files generated with the object to which the annotation is attached.

Tip: example:

```
@cpp(src="something1.cpp,something2.cpp")
atom type SomeAtom()
  ...
end
```

5.7.2 `@cpp(obj="<file-list>")`

- **scope** : package definition, any type definition
- **argument** : comma separated list of file names
- **role** [the files specified as argument will be inserted in the file list] of objects to be linked with objects obtained by the compilation of the generated C++ files (obtained from the object to which the annotation is attached).

Important: You will need to give the linker the paths containing your objects files using `--gencpp-ld-L`

Tip: example:

```
@cpp(src="a/path/something1.o")
atom type SomeAtom()
  ...
end
```

5.7.3 `@cpp(include="<file-list>")`

- **scope** : package definition, any type definition
- **argument** : comma separated list of file names
- **role** : each file in the list will trigger an *include* directive (*ie.* `#include <file>` in the corresponding generated code.

Important: The C++ compiler search path must be set accordingly using `--gencpp-cc-I`.

Tip: example:

```
@cpp(include="a/path/something1.hpp,stdio.h")
atom type SomeAtom()
  ...
end
```

5.8 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 the strong asumptions on which BIP is based. This means that even if *it looks ok at execution*, you will most probably get incorrect result with other tools (*eg.* model checking).

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(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:


```
[SEVERE] In /home/to/example/priorities_in_atom.bip:
Cycle found in priorities in Atom type :
    20:
    21:     priority myPrio1 q < r
------------^
    22:     priority myPrio2 r < p
    23:     priority myPrio3 p < q
```

Priorities may be also defined dynamically using guards involving variables. In this case, cycles are checked at run-time. 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
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:

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

Priorities in an atom is a partial order between its internal ports. It is computed from the rules provided by `priority` statements: it is the result of the application of the transitive closure to the rules. We modify the previous example as follows. We add an internal port `r` such that no transition labelled by `r` is enabled during the execution. Instead of using the priority rule `q < p`, we use rules `q < r` and `r < p`. Due to the computation of the transitive closure before the application of priorities, only transition labelled by `p` can be executed, leading to the execution sequence of the previous example (see above). Even if no transition labelled by `r` is enabled, the priority rule `q < p` is automatically deduced from the rules `q < r` and `r < p`.

```
package priorities_in_atom
  port type Port()

  atom type MyAtom()
    data int i
    port Port p(), q(), r()

    place LOOP, NON_REACHABLE

    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; }

    on r from NON_REACHABLE to NON_REACHABLE

    priority myPrio1 q < r
    priority myPrio2 r < p
  end

  compound type Model()
    component MyAtom a()
  end
end
```

Notice that a set of rule 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:


```
@cpp(include="stdio.h,sideeffects.hpp")
package sideeffects
  port type Port_t(int x)

  atom type Atom_t(int x)
    data int id, dat
    export port Port_t ep(dat)
    port Port_t ip(dat), ip2(dat)

    place I,IP,EP
    initial to I do {id = x; dat = 999;}
    on ip2 from I to I provided (wrong_guard_ip(dat) && 0 == 1)
    on ip from I to IP do { printf("id:%d, data:%d\n", id, dat); }
    on ep from I to EP

  end

  connector type LowC_t(Port_t p1, Port_t p2)
  data int d
  export port Port_t ep(d)
  define p1' p2'
  on p1 p2 up { d = 0; wrong_up(p1.x); wrong_up(p2.x); }
  on    p2 up { d = 0; wrong_up(p2.x); }
  on p1    up { d = 0; wrong_up(p1.x); }
  end

  connector type HighC_t(Port_t p1, Port_t p2)
  define p1 p2
  end

  compound type Top()
    component Atom_t c1(1), c2(2), c3(3)
    connector LowC_t lowc(c1.ep, c2.ep)
    connector HighC_t highc(lowc.ep, c3.ep)
  end
end
```

With `sideeffects.hpp` containing:

```
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()`. Even though the transition labeled by `ip2` is never possible, its guard gets executed, and so, internal data is modified. When the transition labeled by `ip` is triggered, we can see that the data has been wrongly modified (no state change should have been made since the initialization of the system):

```
[BIP ENGINE]: initialize components...
[BIP ENGINE]: state #0: 1 interaction and 3 internal ports:
[BIP ENGINE]:   [0] ROOT.highc: ROOT.lowc.ep({x}=0;) ROOT.c3.ep({x}=-1;)
[BIP ENGINE]:   [1] ROOT.c1._iport_decl__ip
[BIP ENGINE]:   [2] ROOT.c2._iport_decl__ip
[BIP ENGINE]:   [3] ROOT.c3._iport_decl__ip
```

```
[BIP ENGINE]:  -> choose [1] ROOT.c2._iport_decl__ip
id:2, data:-1
```

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 (*eg.* 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 obscur messages that usually deal with low level errors where *user friendlyness* is not the main concern.

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

If you get an output similar to:

```
system: somepath/HelloPackage/AT_MyAtomType.cpp:141: BipError&
AT_MyAtomType::updatePortValues(): Assertion '!_iport_decl__aport.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:

```
[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 `XXXXX` 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]
```



```
    on q from LOOP to LOOP
  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 at 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:

```
...
[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`

Wa 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:

```
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
    on q from LOOP to LOOP

    priority myPrio q < p
  end

  compound type Model()
    component MyAtom a()
  end
end
```

```
      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
end
```

Initially, both processes may aquire the resource since places GET1, GET2, RESOURCE are all marked initially. The, one of the two process aquires the resource leading to a state in which place RESOURCE is not marked. This ensure the mutual exclusion between the use of the resource by the two processes: in this state, the other process cannot aquire the resource. Once the resource is released by a process it is blocked at place SYNC1 or SYNC2, and the other process aquire, 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.

```
$ ./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
...
```

7.5 Priorities

7.5.1 Priorities in atoms

The following example is composed of a single atom that can, at each state, either executes a transition labelled by the internal port p, or a transition labelled by the internal port q.

```
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
```


You are probably using data that the compiler can't [de]serialize. 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*:

```
#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
```

```
  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
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 aquires 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(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
```

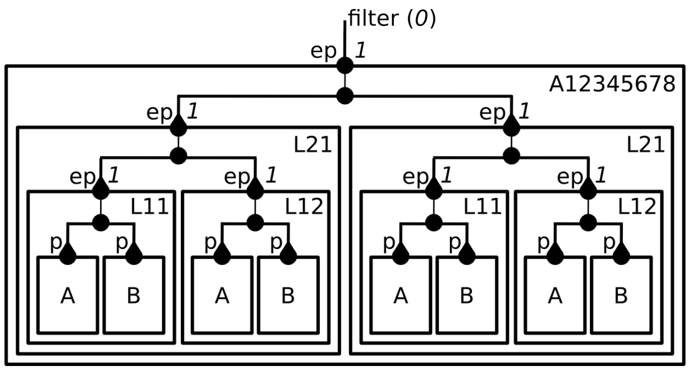


Figure 7.4: Structuring using compounds (names of connectors of type `Plus` are not shown).

```
connector type Plus(HelloPort_t r1, HelloPort_t r2)
  data int number_of_active
  export port HelloPort_t ep(number_of_active)
  define r1' r2'

  on r1 r2
    up { number_of_active = r1.d + r2.d; }
    down { r1.d = number_of_active; r2.d = number_of_active; }

  on r1
    up { number_of_active = r1.d; }
    down { r1.d = number_of_active; }

  on r2
    up { number_of_active = r2.d; }
    down { r2.d = number_of_active; }
end

connector type Filter(HelloPort_t r)
  define r
  on r provided (r.d <= 4) down { r.d = 0; }
end

compound type Layer1(int first)
  component HelloAtom A(first), B(first + 1)

  connector Plus plus12(A.p, B.p)
  export port plus12.ep as ep
end

compound type Layer2(int first)
  component Layer1 L11(first), L12(first + 2)

  connector Plus plus12(L11.ep, L12.ep)
  export port plus12.ep as ep
end
```


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 adding the following in 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
```

6.2.3 Quick-tour of installation

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

```
.
÷-- generic/
|   `-- include
|       ÷-- AtomExportPortItf.hpp
|       ÷-- AtomInternalPortItf.hpp
|       ÷-- AtomItf.hpp
|       ÷-- bip-engineiface-config.hpp
|       ÷-- BipErrorItf.hpp
|
÷-- specific/
|    `-- include
|       ÷-- AtomExportPort.hpp
|       ÷-- Atom.hpp
|       ÷-- AtomInternalPort.hpp
|        ...
`-- lib/
    `-- libengine.a
```

- 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*


```
[BIP ENGINE]:   [58] ROOT.filter: ROOT.plus12345678.ep({d}=4;)
[BIP ENGINE]:   [59] ROOT.filter: ROOT.plus12345678.ep({d}=4;)
[BIP ENGINE]:   [60] ROOT.filter: ROOT.plus12345678.ep({d}=4;)
[BIP ENGINE]:   [61] ROOT.filter: ROOT.plus12345678.ep({d}=4;)
[BIP ENGINE]:   [62] ROOT.filter: ROOT.plus12345678.ep({d}=4;)
[BIP ENGINE]:   [63] ROOT.filter: ROOT.plus12345678.ep({d}=4;)
[BIP ENGINE]:   [64] ROOT.filter: ROOT.plus12345678.ep({d}=4;)
[BIP ENGINE]:   [65] ROOT.filter: ROOT.plus12345678.ep({d}=4;)
[BIP ENGINE]:   [66] ROOT.filter: ROOT.plus12345678.ep({d}=4;)
[BIP ENGINE]:   [67] ROOT.filter: ROOT.plus12345678.ep({d}=4;)
[BIP ENGINE]:   [68] ROOT.filter: ROOT.plus12345678.ep({d}=4;)
[BIP ENGINE]:   [69] ROOT.filter: ROOT.plus12345678.ep({d}=4;)
[BIP ENGINE]: -> choose [21] ROOT.filter: ROOT.plus12345678.ep({d}=4;)
I'm 1, active=0
I'm 5, active=0
I'm 7, active=0
I'm 8, active=0
[BIP ENGINE]: state #1: 1 interaction:
[BIP ENGINE]:   [0] ROOT.filter: ROOT.plus12345678.ep({d}=4;)
[BIP ENGINE]: -> choose [0] ROOT.filter: ROOT.plus12345678.ep({d}=4;)
I'm 2, active=0
I'm 3, active=0
I'm 4, active=0
I'm 6, active=0
[BIP ENGINE]: state #2: deadlock!
```

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 fird 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.

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
```

```
[BIP ENGINE]:   [0] ROOT.filter: ROOT.plus12345678.ep({d}=4;)
[BIP ENGINE]:   [1] ROOT.filter: ROOT.plus12345678.ep({d}=4;)
[BIP ENGINE]:   [2] ROOT.filter: ROOT.plus12345678.ep({d}=4;)
[BIP ENGINE]:   [3] ROOT.filter: ROOT.plus12345678.ep({d}=4;)
[BIP ENGINE]:   [4] ROOT.filter: ROOT.plus12345678.ep({d}=4;)
[BIP ENGINE]:   [5] ROOT.filter: ROOT.plus12345678.ep({d}=4;)
[BIP ENGINE]:   [6] ROOT.filter: ROOT.plus12345678.ep({d}=4;)
[BIP ENGINE]:   [7] ROOT.filter: ROOT.plus12345678.ep({d}=4;)
[BIP ENGINE]:   [8] ROOT.filter: ROOT.plus12345678.ep({d}=4;)
[BIP ENGINE]:   [9] ROOT.filter: ROOT.plus12345678.ep({d}=4;)
[BIP ENGINE]:   [10] ROOT.filter: ROOT.plus12345678.ep({d}=4;)
[BIP ENGINE]:   [11] ROOT.filter: ROOT.plus12345678.ep({d}=4;)
[BIP ENGINE]:   [12] ROOT.filter: ROOT.plus12345678.ep({d}=4;)
[BIP ENGINE]:   [13] ROOT.filter: ROOT.plus12345678.ep({d}=4;)
[BIP ENGINE]:   [14] ROOT.filter: ROOT.plus12345678.ep({d}=4;)
[BIP ENGINE]:   [15] ROOT.filter: ROOT.plus12345678.ep({d}=4;)
[BIP ENGINE]:   [16] ROOT.filter: ROOT.plus12345678.ep({d}=4;)
[BIP ENGINE]:   [17] ROOT.filter: ROOT.plus12345678.ep({d}=4;)
[BIP ENGINE]:   [18] ROOT.filter: ROOT.plus12345678.ep({d}=4;)
[BIP ENGINE]:   [19] ROOT.filter: ROOT.plus12345678.ep({d}=4;)
[BIP ENGINE]:   [20] ROOT.filter: ROOT.plus12345678.ep({d}=4;)
[BIP ENGINE]:   [21] ROOT.filter: ROOT.plus12345678.ep({d}=4;)
[BIP ENGINE]:   [22] ROOT.filter: ROOT.plus12345678.ep({d}=4;)
[BIP ENGINE]:   [23] ROOT.filter: ROOT.plus12345678.ep({d}=4;)
[BIP ENGINE]:   [24] ROOT.filter: ROOT.plus12345678.ep({d}=4;)
[BIP ENGINE]:   [25] ROOT.filter: ROOT.plus12345678.ep({d}=4;)
[BIP ENGINE]:   [26] ROOT.filter: ROOT.plus12345678.ep({d}=4;)
[BIP ENGINE]:   [27] ROOT.filter: ROOT.plus12345678.ep({d}=4;)
[BIP ENGINE]:   [28] ROOT.filter: ROOT.plus12345678.ep({d}=4;)
[BIP ENGINE]:   [29] ROOT.filter: ROOT.plus12345678.ep({d}=4;)
[BIP ENGINE]:   [30] ROOT.filter: ROOT.plus12345678.ep({d}=4;)
[BIP ENGINE]:   [31] ROOT.filter: ROOT.plus12345678.ep({d}=4;)
[BIP ENGINE]:   [32] ROOT.filter: ROOT.plus12345678.ep({d}=4;)
[BIP ENGINE]:   [33] ROOT.filter: ROOT.plus12345678.ep({d}=4;)
[BIP ENGINE]:   [34] ROOT.filter: ROOT.plus12345678.ep({d}=4;)
[BIP ENGINE]:   [35] ROOT.filter: ROOT.plus12345678.ep({d}=4;)
[BIP ENGINE]:   [36] ROOT.filter: ROOT.plus12345678.ep({d}=4;)
[BIP ENGINE]:   [37] ROOT.filter: ROOT.plus12345678.ep({d}=4;)
[BIP ENGINE]:   [38] ROOT.filter: ROOT.plus12345678.ep({d}=4;)
[BIP ENGINE]:   [39] ROOT.filter: ROOT.plus12345678.ep({d}=4;)
[BIP ENGINE]:   [40] ROOT.filter: ROOT.plus12345678.ep({d}=4;)
[BIP ENGINE]:   [41] ROOT.filter: ROOT.plus12345678.ep({d}=4;)
[BIP ENGINE]:   [42] ROOT.filter: ROOT.plus12345678.ep({d}=4;)
[BIP ENGINE]:   [43] ROOT.filter: ROOT.plus12345678.ep({d}=4;)
[BIP ENGINE]:   [44] ROOT.filter: ROOT.plus12345678.ep({d}=4;)
[BIP ENGINE]:   [45] ROOT.filter: ROOT.plus12345678.ep({d}=4;)
[BIP ENGINE]:   [46] ROOT.filter: ROOT.plus12345678.ep({d}=4;)
[BIP ENGINE]:   [47] ROOT.filter: ROOT.plus12345678.ep({d}=4;)
[BIP ENGINE]:   [48] ROOT.filter: ROOT.plus12345678.ep({d}=4;)
[BIP ENGINE]:   [49] ROOT.filter: ROOT.plus12345678.ep({d}=4;)
[BIP ENGINE]:   [50] ROOT.filter: ROOT.plus12345678.ep({d}=4;)
[BIP ENGINE]:   [51] ROOT.filter: ROOT.plus12345678.ep({d}=4;)
[BIP ENGINE]:   [52] ROOT.filter: ROOT.plus12345678.ep({d}=4;)
[BIP ENGINE]:   [53] ROOT.filter: ROOT.plus12345678.ep({d}=4;)
[BIP ENGINE]:   [54] ROOT.filter: ROOT.plus12345678.ep({d}=4;)
[BIP ENGINE]:   [55] ROOT.filter: ROOT.plus12345678.ep({d}=4;)
[BIP ENGINE]:   [56] ROOT.filter: ROOT.plus12345678.ep({d}=4;)
[BIP ENGINE]:   [57] ROOT.filter: ROOT.plus12345678.ep({d}=4;)
```


engine, then you need to provide cmake with *absolutes* paths to engine files: BIP2_ENGINE_GENERIC_DIR and BIP2_ENGINE_SPECIFIC_DIR for the engine interface code (*ie. header* files), and BIP2_ENGINE_LIB_DIR for the compiled engine code.

Example cmake invocation *with* environment variables set:

```
$ cmake ..
-- The C compiler identification is GNU
-- The CXX compiler identification is GNU
-- Check for working C compiler: /usr/bin/gcc
-- Check for working C compiler: /usr/bin/gcc -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Check for working CXX compiler: /usr/bin/c++
-- Check for working CXX compiler: /usr/bin/c++ -- works
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Configuring done
-- Generating done
-- Build files have been written to: /home/a_user/output/build
```

Example cmake invocation *without* environment variables set:

```
$ cmake \
-DBIP2_ENGINE_GENERIC_DIR=/absolute/path/to/engines/BIP-reference-engine-2012.01/include/generic/ \
-DBIP2_ENGINE_SPECIFIC_DIR=/absolute/path/to/engines/BIP-reference-engine-2012.01/include/specific/ \
-DBIP2_ENGINE_LIB_DIR=/absolute/path/to/engines/BIP-reference-engine-2012.01/lib/static \
..
  -- The C compiler identification is GNU
  -- The CXX compiler identification is GNU
  -- Check for working C compiler: /usr/bin/gcc
  -- Check for working C compiler: /usr/bin/gcc -- works
  -- Detecting C compiler ABI info
  -- Detecting C compiler ABI info - done
  -- Check for working CXX compiler: /usr/bin/c++
  -- Check for working CXX compiler: /usr/bin/c++ -- works
  -- Detecting CXX compiler ABI info
  -- Detecting CXX compiler ABI info - done
  -- Configuring done
  -- Generating done
  -- Build files have been written to: /home/a_user/output/build
```

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

```
$ make
```

The result will be a single executable file called `system`.

6.3.2 Running the resulting executable

The resulting executable is called **system** and is created in the build directory created previously (see previous section). It includes both the code generated specifically for the considered BIP2 model and the reference engine. Engines are runtime used for scheduling execution sequences of BIP models.

Once the executable is built, help information is provided when executing **system** with option `--help`:

```
./system --help
Usage: ./system [options]

BIP Engine general options:
 -d, --debug       allows debug of the system, i.e. diplays the state of the system
 --execute         execute a single sequence of interactions (default)
 --explore         compute all possible sequences of interactions
 -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, --verbose     enables the display of the sequence of enabled/chosen interactions (default)
 -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
```

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.*:

```
...
[BIP ENGINE]: initialize components...
[BIP ENGINE]: random scheduling based on seed=6
[BIP ENGINE]: state #0: 14 interactions:
[BIP ENGINE]:   [0] ROOT.f1take1: ROOT.f1.take() ROOT.p1.take_left()
[BIP ENGINE]:   [1] ROOT.f1take2: ROOT.f1.take() ROOT.p7.take_right()
[BIP ENGINE]:   [2] ROOT.f2take1: ROOT.f2.take() ROOT.p2.take_left()
[BIP ENGINE]:   [3] ROOT.f2take2: ROOT.f2.take() ROOT.p1.take_right()
[BIP ENGINE]:   [4] ROOT.f3take1: ROOT.f3.take() ROOT.p3.take_left()
[BIP ENGINE]:   [5] ROOT.f3take2: ROOT.f3.take() ROOT.p2.take_right()
[BIP ENGINE]:   [6] ROOT.f4take1: ROOT.f4.take() ROOT.p4.take_left()
[BIP ENGINE]:   [7] ROOT.f4take2: ROOT.f4.take() ROOT.p3.take_right()
[BIP ENGINE]:   [8] ROOT.f5take1: ROOT.f5.take() ROOT.p5.take_left()
[BIP ENGINE]:   [9] ROOT.f5take2: ROOT.f5.take() ROOT.p4.take_right()
[BIP ENGINE]:   [10] ROOT.f6take1: ROOT.f6.take() ROOT.p6.take_left()
[BIP ENGINE]:   [11] ROOT.f6take2: ROOT.f6.take() ROOT.p5.take_right()
[BIP ENGINE]:   [12] ROOT.f7take1: ROOT.f7.take() ROOT.p7.take_left()
[BIP ENGINE]:   [13] ROOT.f7take2: ROOT.f7.take() ROOT.p6.take_right()
[BIP ENGINE]: -> choose [1] ROOT.f1take2: ROOT.f1.take() ROOT.p7.take_right()
[BIP ENGINE]: state #1: 12 interactions:
[BIP ENGINE]:   [0] ROOT.f2take1: ROOT.f2.take() ROOT.p2.take_left()
[BIP ENGINE]:   [1] ROOT.f2take2: ROOT.f2.take() ROOT.p1.take_right()
[BIP ENGINE]:   [2] ROOT.f3take1: ROOT.f3.take() ROOT.p3.take_left()
[BIP ENGINE]:   [3] ROOT.f3take2: ROOT.f3.take() ROOT.p2.take_right()
[BIP ENGINE]:   [4] ROOT.f4take1: ROOT.f4.take() ROOT.p4.take_left()
[BIP ENGINE]:   [5] ROOT.f4take2: ROOT.f4.take() ROOT.p3.take_right()
[BIP ENGINE]:   [6] ROOT.f5take1: ROOT.f5.take() ROOT.p5.take_left()
[BIP ENGINE]:   [7] ROOT.f5take2: ROOT.f5.take() ROOT.p4.take_right()
[BIP ENGINE]:   [8] ROOT.f6take1: ROOT.f6.take() ROOT.p6.take_left()
[BIP ENGINE]:   [9] ROOT.f6take2: ROOT.f6.take() ROOT.p5.take_right()
```



```
    on r provided (r.d <= 4) down { r.d = 0; }
  end

  compound type HelloCompound()
    component HelloAtom A1(1), A2(2), A3(3), A4(4), A5(5), A6(6), A7(7), A8(8)

    connector Plus plus12(A1.p, A2.p)
    connector Plus plus34(A3.p, A4.p)
    connector Plus plus56(A5.p, A6.p)
    connector Plus plus78(A7.p, A8.p)

    connector Plus plus1234(plus12.ep, plus34.ep)
    connector Plus plus5678(plus56.ep, plus78.ep)

    connector Plus plus12345678(plus1234.ep, plus5678.ep)

    connector Filter filter(plus12345678.ep)
  end
end
```

The behavior of instance of `HelloCompound` is as follows. The first layer of connectors enables interactions ‘`A1.p`‘, ‘`A2.p`‘, ..., ‘`A8.p`‘, ‘`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`‘, ‘`A1.p,A2.p`‘ and interactions ‘`A3.p`‘, ‘`A4.p`‘, ‘`A3.p,A4.p`‘ due to connector `plus1234`, and
- any combination between interactions ‘`A5.p`‘, ‘`A6.p`‘, ‘`A5.p,A6.p`‘ and interactions ‘`A7.p`‘, ‘`A8.p`‘, ‘`A7.p,A8.p`‘ due to connector `plus5678`.

That is, the second layer allows any interaction between a subset of the atoms ‘`A1.p`‘, ..., ‘`A4.p`‘, and any interaction between a subset of the atoms ‘`A5.p`‘, ..., ‘`A8.p`‘. Similarly, the third layer of connectors (*i.e.* `plus12345678`) allows any interaction between a subset the height atoms. This corresponds to a total number of 255 interactions visible from the port `ep` 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 guard 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 exaclty four atoms. Once such an interaction is chosen an executed, the integer value associated to the port `ep` 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==1` 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 BIP2, 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.

```
[BIP ENGINE]: initialize components...
[BIP ENGINE]: state #0: 70 interactions:
```

interactions `AI.p`, `AJ.p` and `AI.p,AJ.p`, and exports the number of atoms participating to the interaction through its port `ep`.

The second layer connects the connectors of the firt layer two by two, that is, `plus1234` connects `plus12` and `plus34`, and `plus5678` connects `plus56` and `plus78`. Since `plus1234` (resp. `plus5678`) is also of type `Plus`, and exports the number of atoms participating to the interaction through its port `ep`.

The first layer consists of a single connector `plus12345678` of type `Plus` connecting the connectors of the previous layer (*i.e.* `plus1234` and `plus5678`), and exporting the number of atoms participating to the interaction.

The last layer is the connector `filter` of type `Filter`, connecting the exported port of the connector of the previous layer (*i.e.* `plus12345678`). It has a guard that allows the interaction `plus12345678.ep` only if the value visible through the port `ep` of `plus12345678` is less or equals than `4`, and it set this value to zero as the interaction `plus12345678.ep` is executed:

```
@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

  connector type Plus(HelloPort_t r1, HelloPort_t r2)
    data int number_of_active
    export port HelloPort_t ep(number_of_active)
    define r1' r2'

    on r1 r2
      up { number_of_active = r1.d + r2.d; }
      down {
        r1.d = number_of_active;
        r2.d = number_of_active;
      }

    on r1
      up { number_of_active = r1.d; }
      down { r1.d = number_of_active; }

    on r2
      up { number_of_active = r2.d; }
      down { r2.d = number_of_active; }
  end

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



```
[BIP ENGINE]:   [10] ROOT.f7take1: ROOT.f7.take() ROOT.p7.take_left()
[BIP ENGINE]:   [11] ROOT.f7take2: ROOT.f7.take() ROOT.p6.take_right()
[BIP ENGINE]: -> choose [11] ROOT.f7take2: ROOT.f7.take() ROOT.p6.take_right()
...
[BIP ENGINE]: state #26: 2 interactions:
[BIP ENGINE]:   [0] ROOT.f7take1: ROOT.f7.take() ROOT.p7.take_left()
[BIP ENGINE]:   [1] ROOT.f7take2: ROOT.f7.take() ROOT.p6.take_right()
[BIP ENGINE]: -> choose [1] ROOT.f7take2: ROOT.f7.take() ROOT.p6.take_right()
[BIP ENGINE]: state #27: deadlock!
```

Interactions or internal ports are chosen randomly amongst the enabled ones. The reference engine is based on a uniform distribution of probability for the choice of the interactions or internal ports. By default, the seed used to initialize randomize choices is computed from the current value of time, but it can be set to a given value using option `--seed`. The execution is stopped if no interaction and no internal port is enabled, or if ctrl-D is hit.

Exhaustive execution

Option `--explore` allows the exhaustive execution of the sequences defined by a model. In order to perform back-tracking, this mode of execution requires the generation of additional code, which is enforced using option `--gencpp-enable-marshalling` when compiling the model.

Important: Enabling option `--gencpp-enable-marshalling` generates code for storing and retrieving states of atomic components, which requires storing / retrieving their variables. For custom types, such code has to be provided by the user (as the definition of the type). For a custom type `custom_t`, the generated code expect the presence of an implementation for following methods:

- `size_t custom_t_sizeof(const custom_t &v)`: returns the number of bytes necessary to allocate for storing the current value of variable `v` of type `custom_t` provided as a parameter. Notice that it can return non constant numbers of bytes that depend on the value of `v` (e.g. useful for a `string`, a list, etc.).
- `void custom_t_toBytes(char *b, const custom_t *ptr_v)` encode the value of a variable of type `custom_t` pointed by `ptr_v` into a sequence of `n` bytes which are stored in a location starting from `b`. The number of bytes `n` must satisfy `n = custom_t_sizeof(*ptr_v)`.
- `void custom_t_fromBytes(custom_t *ptr_v, const char *b)`: decode a sequence of bytes stored at `b` and assign the corresponding value to the variable of type `custom_t` pointed by `ptr_v`. Notice that this method must be able to guess the number of bytes to read from the sequence itself, i.e. it is the user responsibility to provide a way for kwowing when to stop reading bytes from `b`.

Notice that the exploration mode requires comparison of states. It assumes deterministic execution of the above methods, that is, they must provide the same results for the same input values. Obviously, the application of `fromBytes` to the sequence of bytes computed by `toBytes` for a variable `v` must assign to `v` the value it had when calling `toBytes`.

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, *e.g.*:

```
$ ./system --explore
...
[BIP ENGINE]: initialize components...
[BIP ENGINE]: computing reachable states:.............................................
..............................................................................
..............................................................................
..............................................................................
..............................................................................
...
................... 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, proceeds 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 *Optimisation*).

To allow maximal optimization, combine the following:

- pass `--gencpp-optim 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 is 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` add the option `--gencpp-ld-l pthread` when invoking the BIP compiler.

The options proposed by the multithread engine are listed below:

```
BIP Engine general options:
                  (i.e. executes interactions in parallel, if obs. equivalent)
 -d, --debug        allows debug of the system, i.e. diplays the state of the system
 -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
 --threads NB       set the number of threads (by default, use the maximal HW
                    parallelism or 8)
 -s, --silent       disables the display of the sequence of enabled/chosen interactions
 -v, --verbose      enables the display of the sequence of enabled/chosen interactions
                    (default)
 -V, --version      displays engine version and exits
```

The multithread engine does not support any exploration mode and can only execute sequences of interactions. It executes components involved in interactions is 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:

follows.

Important: The behavior obtained when encapsulating a subset of components and connectors into a compound component can be different from the one of the original model if guards are defined in connectors. This is due to the fact that when the port of a connector is exported at the interface of a compound, the priorities are applied to the set of interactions of the connector, that is, only the maximal interactions are visible from the port of the compound.

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

```
ROOT.rcvrs.p({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" :
    34:
    35:     on r1 r2 r3 down { r1.d = d; r2.d = d; r3.d = d; }
------------^
    36:     on r1 r2    down { r1.d = d; r2.d = d;           }
    37:     on r1    r3 down { r1.d = d;           r3.d = d; }
```

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`, ..., `A8` that can execute only if they are *active*, that is, if their integer variable `active` equals to `1`. They are initially active.

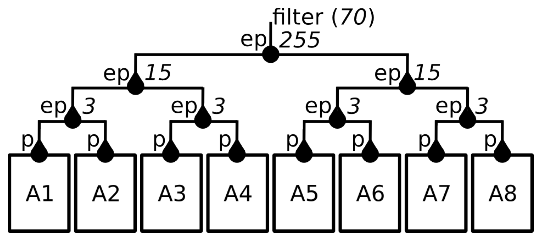


Figure 7.3: Structure of the model: 8 atoms, 4 levels of connectors (names of connectors of type `Plus` are not shown).

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

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
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!
```

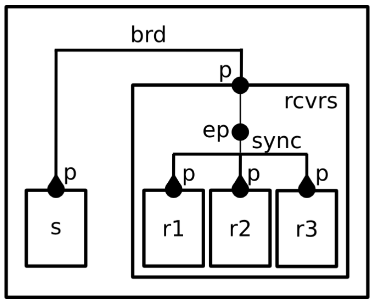


Figure 7.2: Structure of an instance of HelloCompound.

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

- 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
    linked by target "system" in directory ..../output
```

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 !

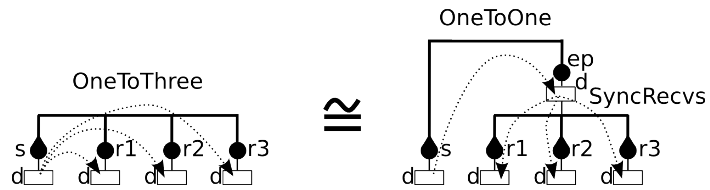


Figure 7.1: Broadcast from `s` using a single connector (left) or a hierarchical connector (right).

```
@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;           }
    on r1    r3 down { r1.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
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:

```
...
[BIP ENGINE]: initialize components...
[BIP ENGINE]: state #0: 1 interaction:
[BIP ENGINE]:   [0] ROOT.brd: ROOT.s.p({d}=0;) ROOT.sync.ep({d}=135026452;)
[BIP ENGINE]: -> choose [0] ROOT.brd: ROOT.s.p({d}=0;) ROOT.sync.ep({d}=135026452;)
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!
```

```
    export port HelloPort_t p(myd)

    place START,END

    initial to START

    on p from START to END
      provided (id == 1 || id == 3)
      do { printf("I'm %d, Hello World received from %d\n", id, myd); }
  end

  connector type OneToThree(HelloPort_t s, HelloPort_t r1, HelloPort_t r2, HelloPort_t r3)
    define s' r1 r2 r3

    on s r1 r2 r3 down { r1.d = s.d; r2.d = s.d; r3.d = s.d; }
    on s r1 r2    down { r1.d = s.d; r2.d = s.d;             }
    on s r1    r3 down { r1.d = s.d;             r3.d = s.d; }
    on s    r2 r3 down {             r2.d = s.d; r3.d = s.d; }
    on s r1       down { r1.d = s.d;                         }
    on s    r2    down {             r2.d = s.d;             }
    on s       r3 down {                         r3.d = s.d; }
  end

  compound type HelloCompound()
    component HelloSender s(0)
    component HelloReceiver r1(1), r2(2), r3(3)
    connector OneToThree brd(s.p, r1.p, r2.p, r3.p)
  end
end
```

In the connector type `OneToThree`, the port `s` corresponding to the sender is a trigger, that is, it can executes alone without synchronizing with the other components. Since other ports are synchrons, `OneToThree` defines the following interactions: '`s`', '`s,r1`', '`s,r2`', '`s,r3`', '`s,r1,r2`', '`s,r1,r3`', '`s,r2,r3`', and '`s,r1,r2,r3`'.

To implement the broadcast of data from port `s`, we use a list of `on` statements that provide `down` blocks of code for all the interactions involving at least one receiver. Notice that even if the interaction '`s`' is not included in this list, it is still considered as a possible interaction, but no transfer of data occurs when '`s`' executes alone.

Due to the guard of the transition labelled by `sync` in the receivers, the interactions enabled after the execution of initial transitions are the following: '`s`', '`s,r1`', '`s,r3`', and '`s,r1,r3`'. As explained in *Priorities*, the application of maximal progress (the default priority rules of BIP2) leads to the execution of the maximal interaction '`s,r1,r3`':

```
...
[BIP ENGINE]: initialize components...
[BIP ENGINE]: state #0: 1 interaction:
[BIP ENGINE]:   [0] ROOT.brd: ROOT.s.p({d}=0;) ROOT.r1.p({d}=0;) ROOT.r3.p({d}=0;)
[BIP ENGINE]: -> choose [0] ROOT.brd: ROOT.s.p({d}=0;) ROOT.r1.p({d}=0;) ROOT.r3.p({d}=0;)
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!
```

We can obtain an equivalent behavior using a hierarchical connector. In this case, receivers are synchronized using a connector `sync` of type `SyncReceivers`. `sync` allows any subset of receivers to participate to the broadcast. A hierarchical connector is build on top of `sync`. For this, we add a broabcast between the sender and the exported port of `sync`. In the block of code provided below we omitted the definitions of types `HelloPort_t`, `HelloSender` and `HelloReceiver` since there are identical to the previous example.

CHAPTER

SEVEN

TUTORIAL

The following sections show how to use BIP on very simple examples. The first part presents general BIP recipes for commonly used patterns. The second part shows more precisely how to interface BIP code with external C++ code with running examples using the reference engine.

Important: All examples in this chapter are available online : http://www-verimag.imag.fr/TOOLS/DCS/bip/examples.tar.gz. Every example contains a `build.sh` script that can be used to compile the example. A master `build_all.sh` is also provided to compile all examples.

7.1 Hello world

This example will be the starting point for all other examples. In a file called `HelloPackage.bip`, write the following BIP code:

```
package HelloPackage
  port type HelloPort_t()

  atom type HelloAtom()
    port HelloPort_t p()
    place START,END
    initial to START
    on p from START to END
  end

  compound type HelloCompound()
    component HelloAtom c1()
  end
end
```

This package contains 3 types:

- 1 port type `HelloPort_t` with no data parameter;
- 1 atom type `HelloAtom` with:
  - 1 internal port declaration `p` of type `HelloPort_t`;
  - 2 places: `START`, which is also the initial place, and `END`;
  - 1 transition labeled by `p` from `START` to `END`
- 1 compound type `HelloCompound` with:
  - 1 component declaration `c1` of type `HelloAtom`.

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 internal port:
[BIP ENGINE]:   [0] ROOT.c1._iport_decl__p
[BIP ENGINE]:  -> choose [0] ROOT.c1._iport_decl__p
[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 `HelloAtom` 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)
```



```
    connector ThreeRendezVous connect(c1.p, c2.p, c3.p)
  end
end
```

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 do { myd = id; }

    on p from START to END
      do { printf("I'm %d, sending Hello World....\n", myd); }
  end

  atom type HelloReceiver(int id)
    data int myd
```