ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

SEMESTER PROJECT

BACHELOR IN COMMUNICATION SYSTEMS

Bidirectional transformation between BIP and SysML for visualisation and editing

Authors:
Clément NUSSBAUMER
Leandro KIELIGER

Supervisors:
Dr. Simon BLIUDZE
Dr. Anton IVANOV

January 13, 2017

Contents

1 Introduction

1.1 Designing software and advantages of model-based design

When it comes to designing software, the classic procedure consists of writing code that complies with the projects requirements first, and then to extensively test for bugs and defects. Although this approach is valid and can produce correct results, it is both time consuming and error prone. Moreover, depending on the time of detection, errors can turn out to be very difficult to correct. Model-based design however allows for verification prior to the coding task. The component-based approach allows to master complexity by structuring systems hierarchically into sub-components where each of them can be designed and validated individually. Despite this approach leading to an increased initial cost of development, it has the significant advantage of detecting and correcting problems early in the life cycle of a project, when errors are relatively inexpensive to fix. Using the BIP framework and its semantics, it is possible to develop models which are correct by construction. This further reduces testing and validation costs.

1.2 Link between the visualisation and the code

Visualisation of systems helps engineers better detect design problems, as well as better conceptualize the system as a whole. It facilitates the communication between the teams involved in a project. Indeed, each subsystem can be displayed at any level of detail, thus efficiently encapsulating complexity when working on different parts of the system. Model-based design creates by definition a well-defined structure which is perfectly suited for visual representations. Each component of the model along with its interactions with the rest of the system be can represented by entities in a diagram.

Of course, visual representations are far beyond the abstraction level usually employed to program a computer. Therefore, to simulate, verify or deploy a model on hardware, a set of instructions needs to be generated from a given programming language. Because of the inherent structure of the model, it is natural to express the need of modifying the system directly from its visual representation, rather than modifying the underlying code.

This observation leads us to a central feature required for most component-based frameworks: the need of having a direct and consistent link between the visual representation of a model and its underlying code. Changes should be bidirectional, this means that any alteration made to the visual representation should be propagated accordingly to the code, and vice versa.

1.3 Our work and structure of the report

In this project we experimented the possibility of visually expressing the structure of a BIP model in a modeling language. We call this conversion the *forward* transformation while modification of the BIP code upon changes made to the SysML model is called the *backward* transformation.

The modeling language chosen to express the BIP semantics is the "Unified Modeling Language" (UML) extension for systems modeling: SysML. It is strictly speaking a profile for UML and is one of the industry standards for designing systems. This choice offers us in the scope of this project the ability to leverage powerful modeling tools designed to work with SysML such as NoMagic's "MagicDraw".

In this report we will start by briefly providing in section 2 the necessary background on the BIP and SysML languages needed to understand how the transformation works. We will continue by offering an overview of the tools we used in section 4. Sections 5, 6 and 7 will explain in details

how the transformation is implemented and how to use the developed tools in order to apply the transformation algorithm. We will finish this report by studying 2 transformations in section 8; one, done step-by-step, for a relatively simple BIP model and one, more complex, which contains nested components and which was used in the context of the CubETH project. [1], [2]

2 Background

2.1 BIP model and tool set

Using the definition from the Verimag laboratory which is in charge of the BIP project [3]:

> *The BIP (Behavior / Interaction / Priority) framework is intended for design and analysis of complex, heterogeneous embedded applications. BIP is a highly expressive, component-based framework with rigorous semantics. It allows the construction of complex, hierarchically structured models from atomic components.*

In the BIP language, a system is defined in terms of its internal components. Every component expresses a behavior and can be connected to other components of the system via connectors. Those connectors define what we call the "interactions" between components of the system. Finally, the BIP language uses priorities to reduce non-determinism, when a choice has to be made between multiple possible interactions.

The BIP language is specified by an ECore meta-model which is used for parsing and generating code: first, the BIP parser creates an instance of the meta-model which corresponds to the entities declared in the code [4]. Then, this model instance is used by the compiler for generating code in a common programming language (C++ by default). We will see later in this report how a BIP Ecore model instance can be used to generate a visual representation of a BIP system.

About ECore

ECore is the base meta-model for the Eclipse Modeling Framework (EMF), a project initially developed by IBM and then transferred to the Eclipse Foundation, which provides code generation and model manipulation tools.

2.1.1 Atoms

The Atom is the most basic component in BIP. It has an internal behavior which is defined by a finite state automaton. An atom can declare a list of ports which allow interactions with neighboring components in the system as well as a list of internal variables.
Ports in an atom have a type, a name and an optional list of previously declared variables that can be exported. This signifies that they become accessible outside the atom itself. An example of a port type declaration can be found in section 8.1.

2.1.2 Connectors

A connector is a stateless element that defines how components in the system interact with each other. They can enforce strong synchronization between a subset of the components they connect but also transfer data between them. A connector can declare an exported port which is unique and can in turn be connected to another component. When a connector connects ports exported by other connectors, it is said to be *hierarchical*.

2.1.3 Compounds

A compound is also a component, but unlike an atom, it does not define its own behavior. Instead, it can contain other components such as atoms and compounds, as well as a set of connectors wiring those components together. Equivalently to the atom, it can declare a set of exported ports that become visible to other components in the model. These ports can reference ports exported either by inner components or inner connectors.

2.2 SysML model and tool set

According to the definition of the Object Management Group (OMG) [5]:

> *SysML is a general-purpose graphical modeling language for specifying, analyzing, designing, and verifying complex systems.*

SysML is an extension[1] of the more general "Unified Modeling Language" (UML), which provides engineers a way of conceptualizing products in terms of subsystems. It allows them to specify the requirements, the behaviors and the constraints for each component of the system in order to ensure the product's effective operation.

About the Object Management Group

OMG is a nonprofit international technology standards consortium. They are at the base of the UML and SysML specifications. [6]

Using SysML, it is possible to create a model representation synthesizing all engineering aspects of a product. It allows a nested model representation of a system, which is used to describe the model at various levels of abstraction: the deeper sub-models cover a tiny part of the product's operation but in great details, whereas the higher-level components link all the elements together while hiding their complexity.

SysML defines nine diagram types, out of which two were used in the scope of this project. Each diagram type specify how SysML entities are visualized as graphical symbols. There are three main categories of diagrams, where each one helps characterizing an aspect of the system:

- Requirements diagram
- Behavior diagrams:

  Activity diagram: behavior based on flow and transformation of input to outputs

  Sequence diagram: behavior based on message exchange between components

  State machine diagram: behavior based on a current state and its transitions

  Use case diagram
- Structure diagrams:

  Block Definition Diagram: describe hierarchy and associations between components

  Internal Block Diagram: shows the internal structure and wiring of a component

  Parametric Diagram: shows constraints that components should respect

  Package Diagram: organisation of the project in terms of packages

In the next subsections we will describe the main components from SysML that were used in the scope of this project.

[1] We explain what we mean by "extension" in section 2.2.1

2.2.1 Profiles

A profile is a mechanism for extending and customizing UML for a particular domain. It contains a collection of stereotypes (see section 2.2.2, below), properties (which can be referred to as tag definitions) and constraints. SysML is an example of such extension; it was designed to allow systems modeling in contrast to UML which is more software oriented.

When modeling the structure of a project, it is necessary to apply a profile in order to use customized elements. A model with a profile applied to it is said to be *profiled*. It is possible to apply multiple profiles to a single model and it is in fact what was realized for this project; we applied to a single model created from a BIP package both the SysML profile and an additional profile we designed specifically for BIP (see section 3.2). Once a model is profiled, it is possible to apply the stereotypes defined by the profile to the model elements.

2.2.2 Stereotypes

A stereotype is a way of defining new elements that are specific to a given domain through the extension of an existing UML metaclass. For example the SysML stereotype ≪ProxyPort≫ extends the UML metaclass "Port" and the ≪Block≫ stereotype extends the "Class" metaclass. Stereotypes may have properties which help define a customized terminology for a specific domain. We describe in the next section which SysML stereotypes were useful in the context of this project.

Note

Stereotypes are usually drawn in diagrams as being surrounded by French quotation marks: ≪≫ . Since we juggle various terminologies in this report, we keep this convention to avoid confusion with other types and names.

2.2.3 Blocks and properties

A block is a discipline-independent element and as the name suggests, it is the building block for modeling systems. Blocks can be assembled to form complex architectures that represent how different elements in the system co-exist. A block can contain properties which are structural features. Those properties have a type (which can be another block or a primitive type such as an `Integer`) that defines their characteristics. Properties are assembled to stipulate the behavior of the block. There exist three kinds of properties which are distinguishable from each other depending on the entity used to type them:

- Part properties: they are typed by blocks and decompose their owning block into constituent elements thus defining a composite relationship.
- Reference properties: they are typed by blocks but do not have a composite relationship with their owning block. Instead, they refer to parts of other blocks.
- Value properties: they are typed by value types and represent quantifiable characteristics of a block.

For example, we can create a SysML block describing a car and another block describing a wheel. The car block can then have four part properties (or simply parts) which are typed by the wheel block to describe the composition relationship existing between a car and a set of wheels.

2.2.4 Ports

A port represents an access part on the boundary of a block or any part or reference typed by that block. They can be connected to each other using binary connectors to support interactions between them. SysML defines two types of ports; full ports and proxy ports. Full ports constitute integral parts of their owning block that can cross the boundary of the block and access external features. They are typed by a block and can have their own behavior as well as the ability of modifying their inputs and outputs. Proxy ports on the other hand are not a part of their parent block. Instead, they provide external access to the feature of the parent block, without having any effect on the inputs or outputs. Proxy ports are typed by an "Interface Block" which specifies the features that can be accessed through the port.

2.2.5 Connectors

Connectors are used to connect two parts inside a block and provide an opportunity for those parts to interact. However, the connector itself does not say anything about the nature of the interaction. The latter is specified by the behavior of the parts it connects and can consist of a flow of inputs and outputs, the invocation of a service in one of the parts or an exchange of messages.

2.2.6 Internal Block Diagram

An internal block diagram is used to show the connections and relationships between parts owned by that block. The boundaries of an internal block diagram always represent a block which is the parent of all entities shown on the internal block diagram.

3 Representation of BIP models in SysML

We leverage the fact that BIP is defined by an Ecore meta-model in order to perform the transformation. Indeed, when compiling a project, the BIP compiler will generate an instance of the BIP Ecore meta-model. It is then possible to intercept the compilation at this stage and directly work with the Ecore model representation. Conversely, we can generate BIP code from the Ecore model instance.

This meta-model instance will be the "source" representation for BIP models. The "destination" will be an instance of a UML model with the SysML profile and the BIP profile (see section 3.2.1 applied to it. The transformation we implemented is then actually performed between these two model representations. The reason for this is that on one hand the BIP compiler can be used to generate BIP code from the Ecore model instance and on the other hand, standard tools for SysML can easily load the profiled UML model.

3.1 BIP $\leftrightarrow$ SysML conceptual mapping and transformation challenges

3.1.1 Representing atoms

We expressed BIP atoms in SysML using the $\ll$Block$\gg$ stereotype. Exported ports are expressed as SysML proxy ports, as they are passive entities that only offer access to the atom behavior and cannot modify the (optional) data they export. (See subsection 2.2.4)

It is quite straightforward to transform the behavior of a BIP atom into SysML as both domains use automata to this end.

3.1.2 Representing port types

The equivalent to a BIP port type in SysML is the $\ll$InterfaceBlock$\gg$ stereotype. Interface blocks are a special kind of block that does not contain any internal structure and are therefore well suited for typing ports.

3.1.3 Representing connectors

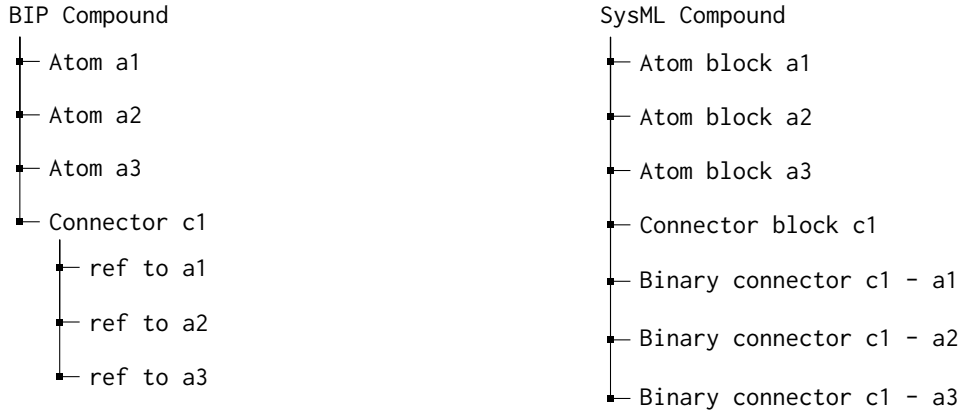
The main difference between BIP and SysML is the fact that SysML connectors are binary and cannot affect the parts they connect whereas BIP connectors can connect multiple components and transfer data between them. To overcome this limitation in SysML, we decided to model BIP connectors by declaring a "Block" for each connector type. We add to this block one port per component to connect and wire them to all components referenced in the BIP connector. This choice of representation has the notable advantage of allowing the SysML connector block to have an effect on the data it carries, just like in BIP. However, as this goes beyond the pure structural aspect of a BIP package that we intended to translate, interactions and data transfers transformations were not implemented in the context of this project, although the structure we chose to put in place would permit it.

3.1.4 Representing compounds

BIP compounds are also expressed using the $\ll$Block$\gg$ stereotype. Indeed, in SysML, blocks can contain other components just like in BIP. Therefore, BIP components declared inside the compound are modeled as `Part Properties`. There is nevertheless one subtlety that arises when transforming BIP compounds to SysML blocks.

As said in the previous subsection, SysML connectors are purely binary and we must declare a connector block to be able to replicate the BIP connector structure. This leads to a structural change:

in SysML, binary connectors connecting the connector block to the connected components must be owned by the enclosing compound block whereas in BIP they are owned by the connector declaration itself. The following hierarchical diagram illustrates the structural difference:



BIP compounds have the ability to export ports declared by their internal components. This is translated in SysML by creating a new port owned by the SysML compound block and then connecting this new port to the corresponding port exported by an internal part. As this port is only here to give access to some internal feature, it is typed by the SysML ≪ProxyPort≫ stereotype.

3.2 Customization

The need for customizing UML for BIP arose for two main reasons. Since all SysML components are modeled using blocks, they are displayed in the same fashion in the diagrams and can be difficult to tell apart. Therefore, it is more convenient to have a way of visually differentiating atoms from connectors and compounds. Moreover, when it comes to the backward transformation, we need a way of differentiating the SysML components to translate them back to BIP. This is why we took the decision of extending the possibilities offered by UML. This was executed by first adding customized stereotypes in a new profile and then customizing the way the SysML modeling tool, MagicDraw, displayed components where such stereotypes were applied.

3.2.1 BIP2 profile (UML extension)

As described in section 2.2.1, it is possible to extend the possibilities offered by UML by the means of a *profile*. We therefore devised a `BIP2` profile which contains the stereotypes described below. Note that because of the similarities between stereotype names in the profiles we used for this project, all BIP stereotypes are prefixed with the lowercase letter 'b'.

- bAtom
- bConnector
- bCompound
- bComponentDeclaration
- bConnectorDeclaration
- bSynchronPort
- bTriggerPort
- ...bInternalPort
- ...bExportedPort
- ...bPackage

These stereotypes are intended to be used in addition to SysML stereotypes. For example, an atom translated in SysML will have both ≪Block≫ and ≪bAtom≫ stereotypes applied to it.

In summary, the SysML profile describes to the SysML-compliant tools how to handle the model and the BIP profile helps us identify the different components visually and for the backward transformation.

3.2.2 MagicDraw custom descriptor

Now that all transformed element types are in theory distinguishable from each other, we can define how MagicDraw will display those elements to the user. MagicDraw uses what is called a "descriptor" to specify how different elements should be drawn on a diagram. A descriptor defines colors, shapes and layout properties for each element type and there exists at least one descriptor per diagram type. For our project we customized the "Internal Block Diagram" descriptor in order to express the structure of BIP models.

This custom descriptor can be found in the "ExtendedSysML" folder as explained in section 7.4. It modifies the diagram as follows:

- Connector declarations are drawn as a rectangle with a gray background
- Synchron ports are drawn with a black circle
- Trigger ports are drawn with a black triangle with rounded corners

All other elements are left with their default display properties: a rectangle with a yellow/orange background for component declaration and a gray square with green outline for all ports which are not explicitly typed ≪bSynchronPort≫ or ≪bTriggerPort≫ .

4 Tool overview

4.1 MD Workbench

MD Workbench[7] is an Eclipse extension commercialized by the SODIUS company that enables transformations between various data structures and models (e.g. UML, Ecore, XML, ...). During this project, we used it to read serialized BIP files to generate meta-model instances and then to convert them to SysML.

As explained earlier, SysML is a profile for UML and therefore SysML files generated using MD Workbench are strictly speaking "UML files profiled with SysML". However, for the sake of simplicity, we will call them SysML files in this report.

The UML meta-model is available in MD Workbench as a MD Access plugin that also contains the SysML 1.3 Profile. Therefore, in order to generate SysML files, we need to create a UML Model using MD Workbench's UML meta-model handler, and then to apply the SysML profile onto it.

The serialized BIP files, which are model instances based on the BIP meta-model, are not natively supported by MD Workbench. However MD Workbench allows for the generation of a specialized meta-model handler, called a MD Access plugin. This process is covered in the following subsection.

4.2 MD Access

MD Workbench provides a generic way of creating "MD Access plugins" from an Ecore meta-model. These plugins provide a way of reading from, or writing to, meta-model instances. They enable the MD Workbench software to work with any kind of data structure, thus making it a universal model transformation tool.

Generation of a MD Access plugin is done using the "MD Workbench/Metamodel" import wizard. This wizard first asks the user for the Ecore Meta-model describing the data structure for which to create a MD Access plugin, and it then lets the user select which namespace prefix and which namespace URI to use for the generated MD Access plugin.

Namespace prefix and URI

A namespace is a mean of providing structure and of categorizing elements in an XML/XMI/Ecore/... file. A namespace is specified by a namespace prefix and URI; the namespace prefix can be seen as the folder in which elements belonging to a namespace are located, and the URI is used to uniquely identify the namespace. This is better explained by an example: in the BIP2 meta-model, there are several namespaces: one of them is the `types` namespace. Its namespace prefix is `bip2.ujf.verimag.bip.types`, the associated URI is `http://bip2/ujf/verimag/bip/types/1.0` and it contains the various BIP types mentioned in section 2.1 (e.g. `AtomType`, `ConnectorType`, ...)

The choice was done not to change the default values, hence the BIP2 MD Access namespace prefix is `bip2`, and the associated namespace URI is `http://www.mdworkbench.com/bip2`.

unfortunately, when the BIP MD Access plugin is generated via MD Workbench, the original BIP Ecore meta-model structure is modified, and its package structure is flattened. In the original BIP meta-model we have 11 sub packages (e.g. `types`, `data`, `port`, `behavior`, `priority`, `connector`, ...) each with a specific namespace prefix and URI and each containing specific BIP classes, whereas in the BIP MD Access meta-model, all the BIP classes are located in the same namespace (with the namespace prefix `bip2` and the namespace URI `http://www.mdworkbench.com/bip2`). This package flattening limitation will prevent any use of the BIP MD Access extension to generate BIP models

that are compatible with the BIP compiler, as the models generated with the BIP MD Access plugin will not have the same structure as the models generated using the original BIP meta-model.

4.3 BIP compiler libraries

As the BIP MD Access plugin can not be used to generate BIP model instances, we decided to use the original BIP2 libraries contained in the BIP compiler distributed by Verimag [8] to generate model instances. These Java libraries contain the BIP2 Ecore meta-model with the correct sub-package structure, and we can therefore use them to generate `.xmi` files that will be correctly parsed by the BIP compiler.

4.4 MagicDraw

MagicDraw is a software and system modeling tool developed by NoMagic [10]. It fully supports the UML2 meta-model and is one of the standard tools used in the industry. In the context of this project we used MagicDraw to generate SysML diagrams for the profiled models we exported via our transformation software. Naturally, we also used MagicDraw to modify or even create from scratch SysML models that were later passed as input to the inverse transformation for generating BIP code.

MagicDraw allows for customization at various levels through user-defined modules. We used for instance in this project the ability to change the visual representation of an entity based on its type or its applied stereotypes via a descriptor. (See section 3.2.2)

5 Implementation details

5.1 Preliminary explanations

Since the terminologies used in the meta-models and in the Java code can be quite confusing, we used in the following sections a color code to distinguish different types of elements:

- `Java-related fields and variables are colored in purple`
- `BIP and SysML meta-model types such as BIP AtomType or UML Property are colored in blue`
- `Attributes of meta-model types such as BIP atoms' internalPortDeclarations list or UML ownedAttribute list are colored in orange.`

When transforming BIP entities to SysML, one problem that often arise is to keep track of the created components to access them later when they are referenced from another entity.

For example, suppose that at the beginning of the transformation, we find a BIP `PortType` called "bip_type_A". We create the appropriate SysML `≪InterfaceBlock≫` for it, say "sysml_type_A", and further in the transformation process, we find a BIP `AtomType` which precisely uses a port of type "bip_type_A". When converting the BIP `AtomType`, we will need to set up its port with the right SysML type, that is to say "sysml_type_A".

Therefore, to find this type back, we must keep track of the association between "bip_type_A" and "sysml_type_A" during the whole transformation process. This can be done in Java using HashMaps[2]. For the forward transformation we created the association tables listed below (similar tables were used for the return transformation):

- PortType to Class — `portTypeMap`
- ConnectorPortParameterDeclaration to Port — `portDeclarationMap`
- ComponentType to Class — `componentTypeMap`
- ConnectorType to Class — `connectorTypeMap`
- PortDeclaration to Port — `internalPortMap`
- PortDeclaration to Port — `exportedPortMap`
- ComponentDeclaration to Property — `componentDeclarationToPropertyMap`
- ConnectorDeclaration to Property — `connectorDeclarationToPropertyMap`
- bip2.State to uml21.State — `statesMap`
- String to uml21.DataType — `dataTypeMap`

To keep track of imported BIP packages we store them in the `importedPackagesSet` field which is of type HashSet. See section 5.1.1 for explanations about the utility of such a set.

The following diagram illustrate how our transformation software integrates with the existing tools to perform the model transformation.
Note how the MDAccess plugin offers an interface for manipulating the model instances, except for the return transformation where we directly used the BIP libraries, for the reasons mentioned in section 4.2.

[2]Note that in Java, if an object does not redefine the "equals" and "hashCode" methods then two keys in a HashMap are considered to be the same if and only if they reference the same object in memory. Since the meta-model elements do not redefine those methods, the hash map will create an entry for every BIP entity transformed, which is precisely what we want.

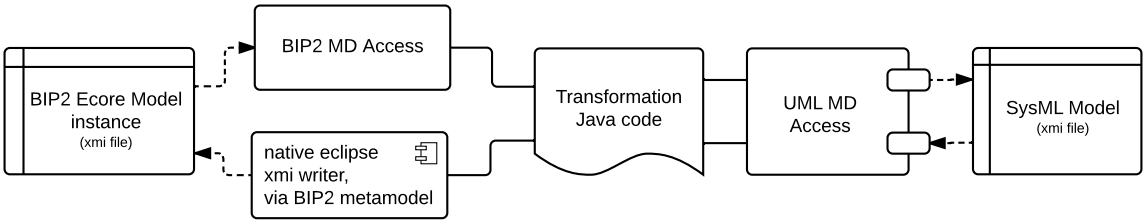


Figure 1: Transformation flow BIP ↔ SysML

5.1.1 Package transformation

The main challenge in converting the BIP packages to SysML is to keep the package structure from one representation to the other. We leveraged the fact that each BIP `Package` references imported packages in its `used_packages` attribute. When we encounter a `Package` import, it means that we first have to transform the imported `Package`, otherwise we may arrive at a point in the transformation where there are references to types that have not yet been transformed. To avoid this, we iterate over the imported packages list and recursively apply the transformation to them before pursuing with the transformation of the "root" package.

Note

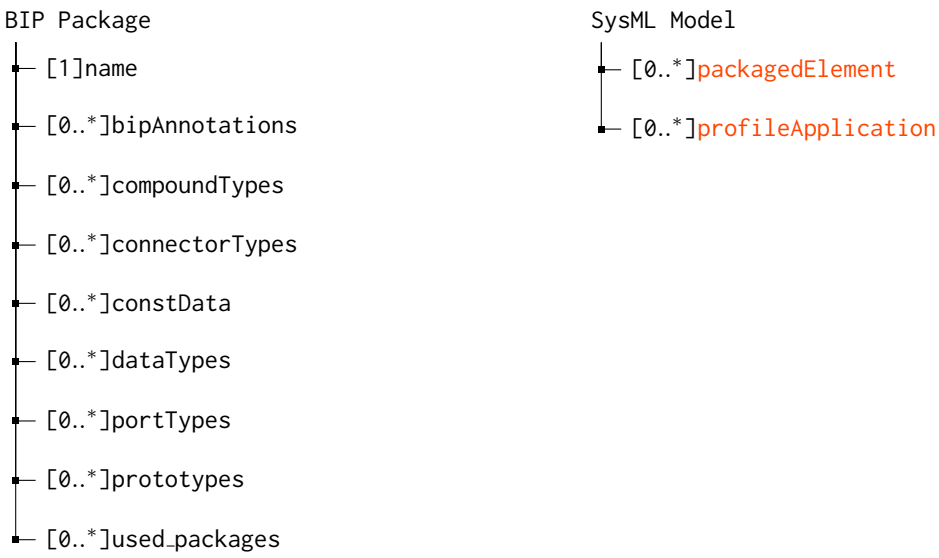
For simplicity purposes, our implementation of the package transformation does not support cyclic imports. That is, if package A uses package B and package B uses package A, our transformation will not halt.

Every SysML package created is added to the `importedPackagesSet` so that a given BIP package is transformed only once even if it is imported in multiples places.

Each BIP package transformation creates its own atom, connector and compound converters. Therefore entities created during a given `Package` transformation will have the corresponding SysML package as their owning package. Association tables however, are global to the model transformation which allows SysML entities registered during the transformation of a `Package` to be retrieved during all subsequent `Package` transformations.

For example, if a `Package` declares every `AtomType` for the project and nothing else, then during the transformation of this package we will have created all the corresponding SysML atom `≪Blocks≫` . The `componentTypeMap` will be populated with BIP `AtomType` - SysML `Class` pairs (We use the `Class` meta-type as it is the supertype of `≪Blocks≫`). Suppose that later on, there is a compound which declares components typed by the previously converted BIP `AtomTypes`. Then, thanks to the association tables, we will be able to retrieve the corresponding SysML atom `≪Blocks≫` type even if the transformation is now happening inside another package.

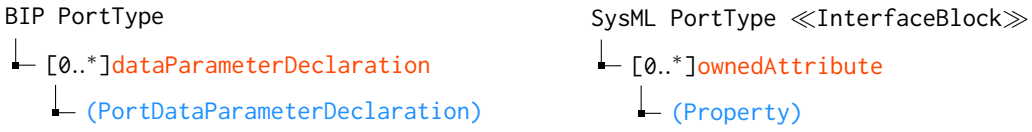
Local converters allied with global association tables allow us to replicate the BIP package structure. Note that the transformation is done in the order formed by the following sub-sections of this report.



5.1.2 Port types transformation

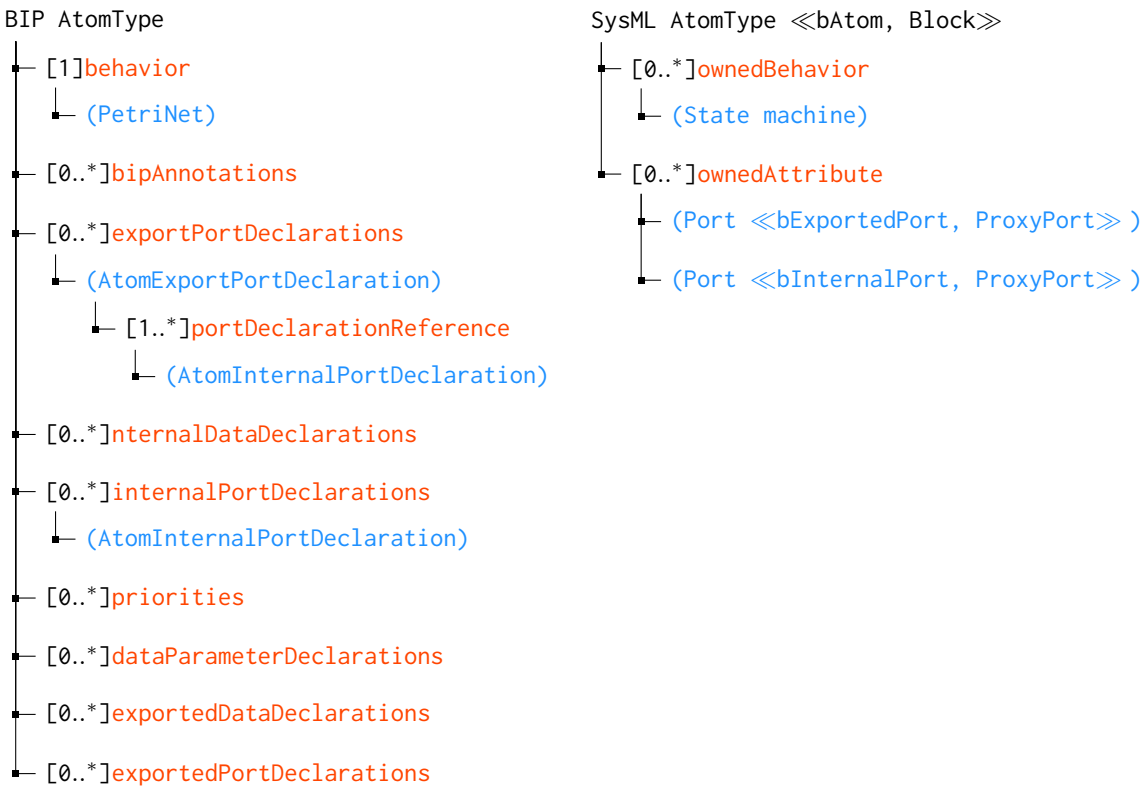
BIP PortTypes declaration are quite simple to transform; they only store a list of PortDataParameterDeclarations. Therefore, to convert a BIP PortType declaration to SysML, we create a new ≪InterfaceBlock≫ and add as owned attributes the specified data parameters which are represented using UML's Property type. We finally register a key-value pair where the BIP port type is the key and the newly created SysML interface block is the value into the portTypeMap association table.

The tree diagrams that you will find in the following sub-sections illustrates on the left-hand side the structure of BIP types as they are organized in the BIP Ecore meta-model, as well as the multiplicity of their attributes. The right-hand side shows the structure of a corresponding SysML model. Meta-model types between parenthesis denote which types of entities could be contained in a given tree node.



5.1.3 Atom type transformation

Note in the following tree diagram how BIP defines exported ports for the AtomTypes: it first defines the internal ports of the atom and then references a subset of those internal ports as being exported. On the SysML structure side, ports are registered as "owned attributes" and the state machine as an "owned behavior".



In the SysML Model we applied the ≪bExportedPort≫ stereotype to the exported ports, and the ≪bInternalPort≫ stereotype to the other ports (i.e. the internal ports).

The BIP behavior attribute, which is a PetriNet, is converted to a SysML StateMachine. The details of the conversion for transitions between states are the following:

- If a BIP port labels a transition, then the corresponding SysML port is marked as a trigger for the SysML transition.
- Transition effects are mapped to SysML OpaqueBehaviors which contain a textual representation of the original BIP Expressions.
- Guards for BIP transitions are mapped to OpaqueExpressions that contain a textual representation of the BIP guards.

The dataParameters, internalDataDeclarations, priorities, exportedPortDeclarations, exportedDataDeclarations and DataParameterDeclarations attributes were not converted to SysML as they do not influence directly the structure of the system.

We keep track of the newly created internal and exported ports by registering a key-value pair where the BIP port is the key and the SysML port is the value in the internalPortMap and exportedPortMap respectively. This allow us to easily retrieve the corresponding SysML ports whenever a reference to the BIP ports shows up in the meta-model instance.

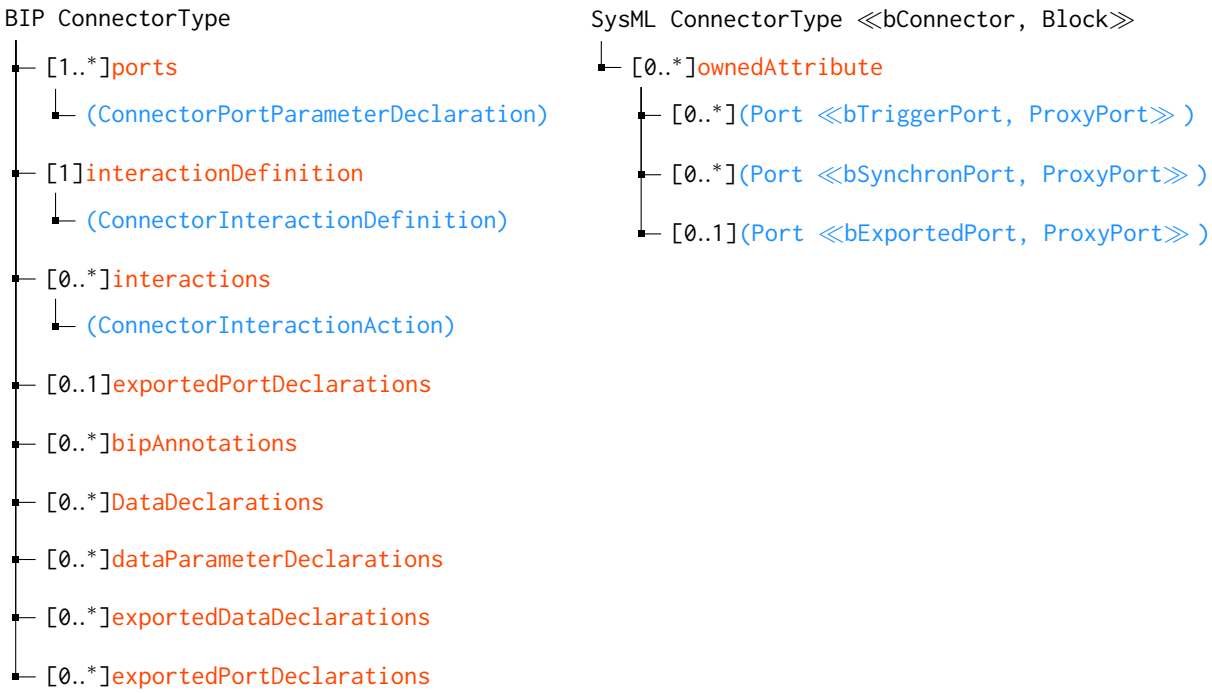
5.1.4 Connector type transformation

The following diagram illustrates the mapping between BIP ConnectorType and SysML connectors, which are in fact SysML ≪Classes≫ typed with the ≪Block≫ and ≪bConnector≫ stereotypes.

The BIP language defines the properties of the ports at multiple places: the ports are declared within the ConnectorPortParameterDeclarations attribute, but the trigger/synchron property is defined in the interactionDefinition attribute, which itself references the aforementioned ConnectorPortParameterDeclaration. In the SysML representation however, we made the choice to simply store the trigger/synchron property of each port in the port definition itself.

There can only be one exported port per BIP connector, and the corresponding SysML port is typed with the ≪bExportedPort≫ stereotype.

Additional information such as the interactions, bipAnnotations and DataDeclarations were not converted.



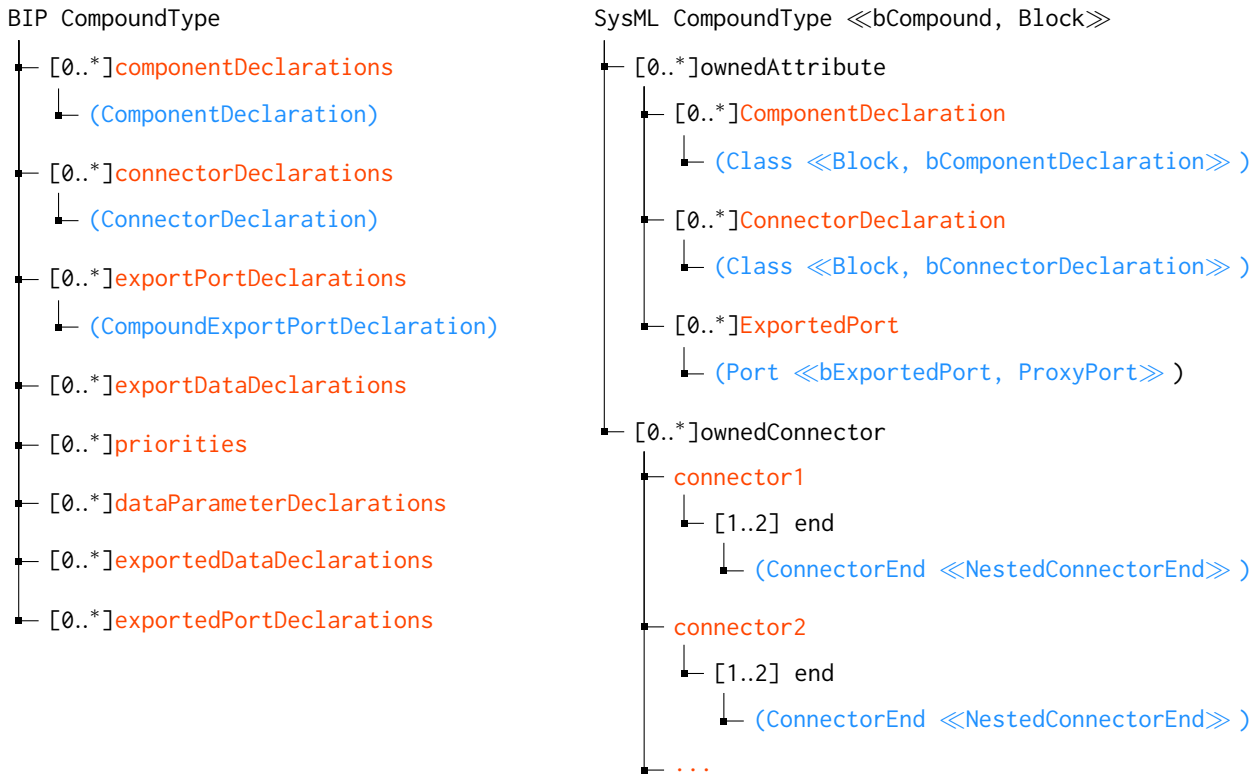
5.1.5 Compound transformation

BIP Compounds are by far the most complex elements to transform. Indeed, they usually employ components whose type were previously declared during the transformation process and they store the actual structure and wiring of the system. As usual, you will find below the structure of a BIP Compound as it is represented in the Ecore meta-model, as well as the target structure we achieved for SysML. The transformation of BIP Compounds is done in four phases:

1. Instantiate the components inside the compound block.
2. Instantiate the connectors and connect components and connectors inside the compound block.

3. Create exported ports declared by the compound.
4. Connect exported ports to the ports of the internal parts they refer to.

Of course, the order in which we apply the different transformation phases matter as it would be impossible to connect components and connectors if there were still missing declarations.



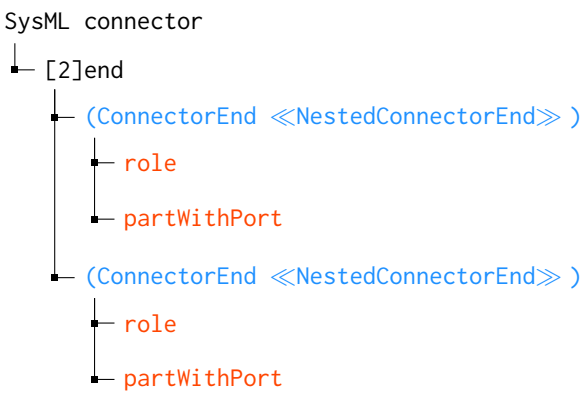
First phase We iterate over the ComponentDeclarations and, for each declaration, create a Property which is stereotyped ≪bComponentDeclaration≫ . This property is added as an ownedAttribute of the compound we are transforming. The Property name is set to match the name used in the BIP model for the ComponentDeclaration. To set the correct type for the SysML property, we need to retrieve the BIP type of the BIP ComponentDeclaration and find the associated SysML type in the componentTypeMap association table. Finally, we register a key-value pair where the BIP ComponentDeclaration is the key and the UML Property is the value into the componentDeclarationToProperty association table. This will be useful for the second phase, when we connect components and connectors.

Second phase It consists of two steps. First, we iterate over the ConnectorDeclaration and, for each declaration, create a Property which is stereotyped ≪bConnectorDeclaration≫ . The procedure is similar to the first phase but this time we query the connectorTypeMap and register the key-value pair into the connectorDeclarationToProperty map.

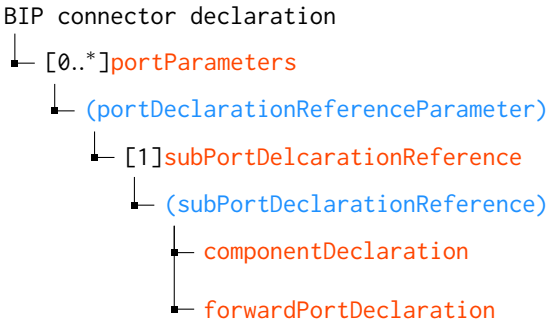
The second step is to actually connect the connector declaration Property to the correct parts. As explained in section 3.1.3, this is done by using SysML binary connectors. Such connectors have two ends (typed ≪NestedConnectorEnd≫), each of which must declare a role and a partWithPort attribute (optional). A role is the port entity defined in an atom or connector type while the partWithPort

is the actual component or connector declaration that owns the port.

The difference between these attributes is better explained by an example: suppose we declare a compound C and an atom type A which has an exported port p. Inside the compound C are declared two components (atoms) typed by A: $comp1$ and $comp2$. To connect them using a SysML connector, the `role` attribute of both ends will be the port p. However, the `partWithPort` attribute will be $comp1$ on one side and $comp2$ on the other.



The information about which part to connect is stored in the BIP `ConnectorDeclaration`. For each port specified in the `ConnectorType`, there is a `PortDeclarationReferenceParameter` in the `ConnectorDeclaration`. This reference holds a `subPortDeclarationReference` attribute which is a grouping of two elements very similar to the SysML attributes for connectors; a `forwardPortDeclaration` which is the equivalent of `role` and a `componentDeclaration` which is the equivalent of the `partWithPort` attribute. We therefore use these attributes to create the SysML connector, as illustrated by the tree structure below:



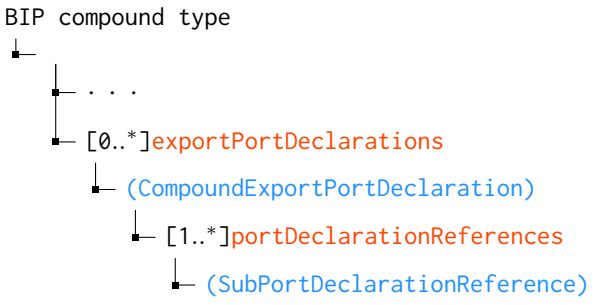
Using this information we are now able to connect connector-component pairs. SysML ports corresponding to BIP ports specified in the `forwardPort` attribute are retrieved using the `exportedPortMap` association table and SyML part properties using either `componentDeclarationToProperty` or the `connectorDeclarationToProperty` depending on the type of the entity at the other end of the connector.

Due to a rendering issue on MagicDraw where connectors do not appear to be connected to the right components, we needed to manually set an attribute specific to the `ConnectorEnd` type. This special

attribute, called propertyPath takes the same value as the partWithPort attribute in the connector end[3].

Third phase This phase is trivial as it only consists of creating exported ports for the compound, based on the exportPortDeclarations of the BIP compound. All SysML compound ports are associated with their respective BIP compound ports into the exportedPortMap.

Fourth phase The last step shares many similarities with the component connecting phase. Each time a compound exports a port from one of its internal components, it creates in the meta-model instance a compoundExportPortDeclaration.



To create the corresponding SysML connector, we set the role attribute in the connector ends as explained previously. However, this time one end of the connector is connected to the compound itself, therefore it does not correspond to any part property inside the compound ≪Block ≫ and the partWithPort attribute should be left empty on this particular side. We can set partWithPort as usual on the opposite side as we know that it will be connected to a component or a connector declaration. Again, the special attribute propertyPath is set manually to ensure a correct representation in MagicDraw.

To conclude the compound transformation we only need to add the newly created SysML compound to the componentTypeMap. Compounds are considered components because they should be usable inside other compounds to define a nested structure.

5.2 Inverse transformation

The backward transformation follows a similar process than what has been described in the previous sections. Therefore, we will not cover its details, as they mainly consist of rebuilding the original BIP structure from the SysML entities. Since SysML models were customized to reflect the BIP nature of their components, it is very easy to retrieve entities by their type and perform the inverse transformation in the same order as the forward transformation. (First the packages, then the port types and so on.)

There exists one main difference nevertheless. Instead of using MD Workbench and MD Access to set up the BIP model from the SysML representation, we directly used the BIP libraries provided by Verimag (as mentioned in section 4.3). This had the immense advantage of providing us with Java factories for creating BIP types. These factories provide complete interfaces for instantiating BIP entities easily while they take care of creating the proper structure and setting the appropriate

[3]It is not clear whether the issue comes from the MD Workbench implementation or if the propertyPath is intended to be set manually

references for the objects in the background.

Ideally, the forward transformation should have been done the same way as explained in section 10.2.2. Unfortunately, because of time constraints this was not possible.

The following snippet shows how a connector factory can be used to create a BIP connector declaration. This connector declaration is then added to its owning compound:

```
ConnectorDeclaration conDecl = ConnectorFactory.eINSTANCE.createConnectorDeclaration();
conDecl.setName(...);
conDecl.setType(...);

   [...]

compoundType.getConnectorDeclarations().add(conDecl);
```

6 Using MagicDraw to generate BIP models

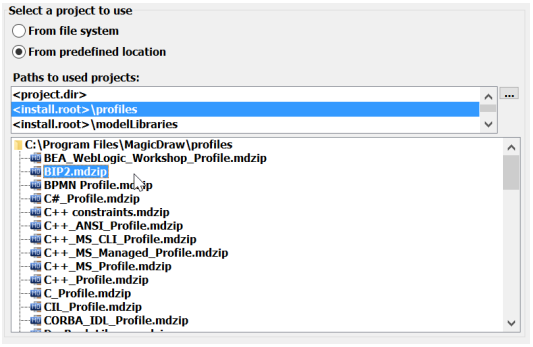


Figure 2: Importing the BIP2 profile

In this section we demonstrate how to model a simple system from scratch using MagicDraw with the objective of generating BIP code from it. We first create a new SysML project that we will call "DFA". DFA stands for Deterministic Finite Automaton, as the system described below is a classical example used in automata theory. Before starting, make sure you configured MagicDraw to use both the custom BIP profile and the BIP descriptor. Instructions for carrying out this operation are given in section 7.4.1 and 7.4.2.

We can now start modeling our system. For this demonstration we will create the structure of a Turing machine capable of determining if its input is a number divisible by 3 or not. This project consists of two atoms and one connector; the tape atom models the input of the Turing machine while the processor models the action table and the transition table. Upon each " tick" from the BIP engine, the connector transfers one bit from the tape to the processor. For this example however, the data transfer, which represent the reading head of the machine, will not be modeled. The input on the tape is supposed to be an integer representing a binary number stored in little-endian format.

Now that we have an idea of what the system consists of, we can start creating SysML entities for it. First, we add two atom types to the root model (DFA). This is done by right-clicking the DFA model and selecting "Create Element" (or Ctrl+Shift+E). We select the BIP atom type as this will create a new UML class with the ≪Block≫ and ≪bAtom≫ stereotypes applied to it. Name the two atom types "Tape" and "Processor" respectively.

Duplicate this procedure to create a new connector type that we will call "ReadingHead".

To allow for data transfers using the exported ports, we create the equivalent of a BIP port type, which is an ≪InterfaceBlock≫ , and add to it a value property "k" of type Integer.

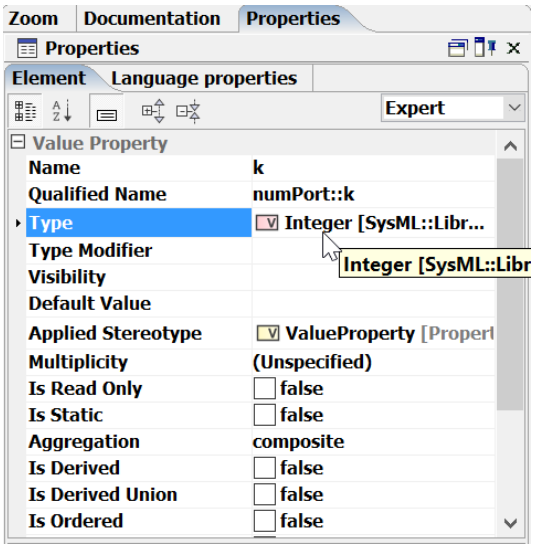


Figure 3: Property panel for Interface Block

Specifying the type of an element can be done via the "Property panel" on the bottom left of the screen. We can now add a root compound that will contain every element of our system. To do so, create a new compound element in the same fashion as previously done for the atoms and name it "EnclosingCompound".

6.1 Populating the root compound

Figure 4 shows what the project structure should look like at this point. We will now start assembling our system using the previously declared entities. Select the "EnclosingCompound" object and press

the “Create diagram” button (or Ctrl+N). We will create a BIP-specific diagram (Section 7.4.2 shows how to import the BIP descriptor), which is a modified version of an “InternalBlockDiagram”. Using the palette on the bottom left of the diagram window, we create two components. Using the “Smart manipulators” which are the buttons that appear when clicking on an element in the diagram, we can specify the type of all three declared inner elements by clicking the red “T” character.

Again using the “Smart manipulators”, we can add one external port for the Tape and the Processor, and two synchron ports to the connector. With the same procedure, we specify the type of each declared port to be the “numPort” ≪InterfaceBlock≫ .

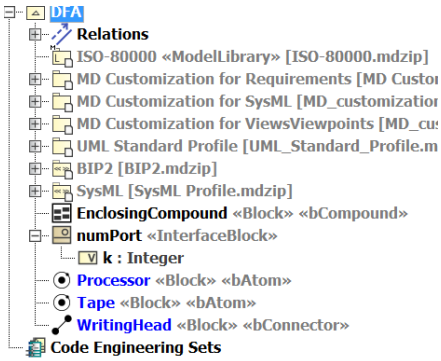
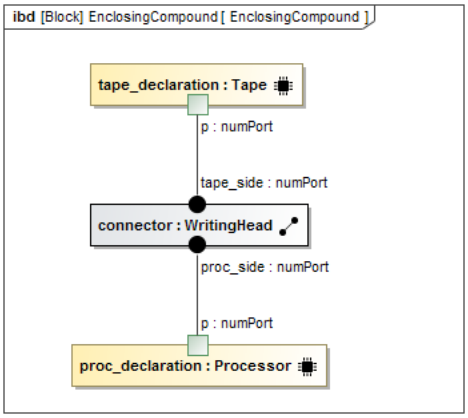


Figure 4: Project containment tree.

Note

Ports added via the internal block diagram are automatically added to the port type as it can be seen on the project containment tree on the left of the screen. However, deleting an element in the diagram does not remove it from the project structure.

Finally, we connect the corresponding ports and press the “Quick Layout Button” to get the following structure:



Tip

As the name suggests, the quick layout button (Ctrl+Q) allows to quickly reorganize the diagram content to improve readability.

6.2 Adding behavior to the atoms

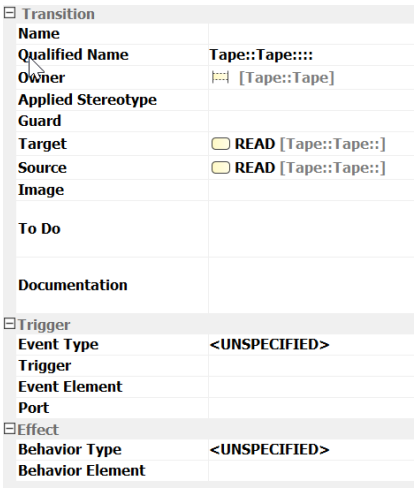


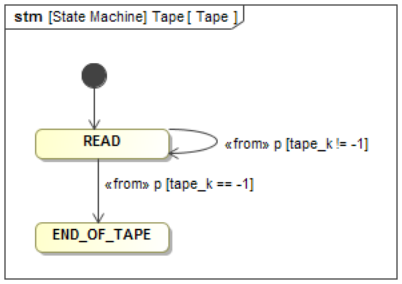
Figure 5: Property panel for transitions. The port field specifies which port enables the transitions, the guard field is self-explanatory

Atom behavior are specified using SysML state machines. To create state machines, simply create a new diagram (Ctrl+N) of type "SysML State Machine Diagram". MagicDraw automatically creates an initial region that we can leave unnamed. Since BIP uses 1-safe automata, we will not need any additional region.

Thanks to the palette on the bottom left of the diagram window, we can add states and transitions to the state machine. We start by defining the Tape's behavior. It needs two states, READ and END_OF_TAPE, READ being the initial state of the machine.

The idea is that from the initial state to READ, we load an integer parameter given when instantiating the Tape atom. This will model the content of the Tape. At each "tick", we advance the tape head by one position by dividing the current number by 10, thus eliminating the rightmost bit (the most significant one). The rightmost bit is sent to the processor atom and the cycle is repeated until the integer number is left equal to zero.

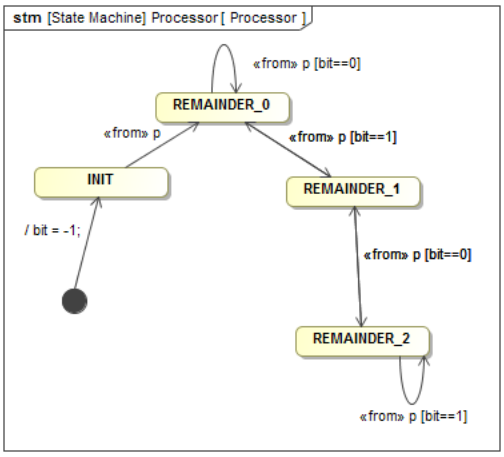
Transitions for the Tape atom are defined as shown in the state machine diagram below. Guards and ports references can be specified using the property panel for a transition instance (see figure 5).



Note

Functions calls and effects can be specified in SysML and will be printed along the transition arrows, but they will not be converted to BIP as our implementation does not handle functional elements.

Similarly, we construct the state machine for the processor atom:



6.3 Global structure visualization

We would now like to visualize the atom behavior directly in the enclosing compound. This is achievable by creating an internal block diagram or BIP diagram for each atom type and then dragging the state machine diagram from the project containment tree into the internal block diagram.

In the BIP diagram for the enclosing compound, right-clicking on the components and selecting "Display internal parts" allows us to visualize the behavior of each atom. (Remember that you can use the quick layout button in case adding the behavior made the diagram unreadable)

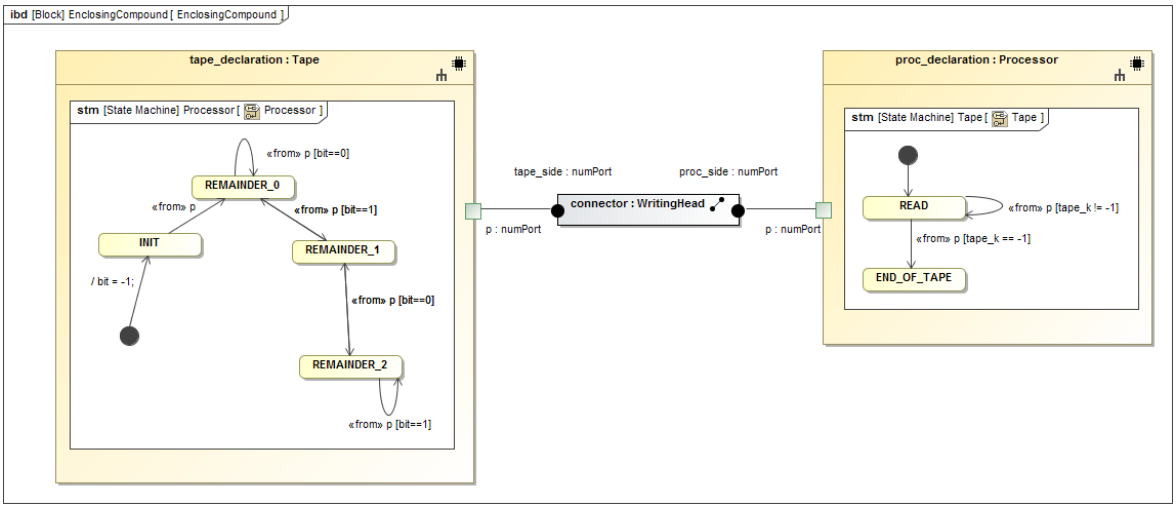


Figure 6: Final structure

6.4 Generating the BIP code

To perform the backward transformation we need to export the project into files comprehensible by our software. This is done via the File menu and the export option. We use the "Eclipse UML2 (v2.X) XMI File" format.

Among the files exported by MagicDraw we find an UML file whose name corresponds to our SysML project name. This is the file that we need to specify for the backward transformation. Upon completion of the transformation, the software will produce as output an XMI file whose name corresponds to the base SysML model; in our case "DFA". The inverse BIP compiler can work on this XMI file to generate the code shown in appendix A.2.

7 Using the developed tools

The transformation tool is distributed as a Java Archive (`jar`) with a Command Line Interface (CLI).

In order to better visualize and work with SysML models, we also developed a set of extensions and settings for NoMagic Inc. "MagicDraw" software.

7.1 Command line interface

The transformation tool is contained in the file `Bip2ToSysML.jar`, and it depends on several libraries that are located in the `modules/` folder. The directory structure of the transformation tool and its dependencies is represented in the following diagram:

```
├─ modules/
│  ├─ bip2MDAccess/
│  │  └─ Contains the BIP2 MD Access plugin, used to read serialized BIP files
│  ├─ bipc—2014_05/
│  │  └─ Contains the BIP Compiler libraries, from which we use the BIP meta-model
│  ├─ eclipse-libs/
│  │  └─ Contains the eclipse libraries from the BIP compiler
│  ├─ sodius/
│  │  └─ Contains SODIUS MD Workbench libraries
│  └─ jopt-simple-6.0.jar
│     └─ A small library used for the command line interface
└─ BIP2ToSysML.jar
```

7.2 Using the CLI to transform models

7.2.1 CLI usage

The CLI is executed with the following command: `java -jar BIP2ToSysML.jar`, to which we append the argument `--mode` to specify the transformation mode.

- `--mode direct` (or shorter `-m d`), when doing a BIP2 → SysML transformation
- `--mode inverse` (or shorter `-m i`), when doing a SysML → BIP2 transformation

The CLI then takes a mandatory `--input-file` argument, that specifies the file that is to be transformed, and an optional `--output-file` argument. The `--output-file` argument specifies the file in which the SysML model is written when doing a direct transformation.

When doing an inverse transformation, `--output-file`, is used to specify the folder in which the converted BIP packages will be written (each BIP package will be written to a file whose name corresponds to the package name). If the `output-file` argument is not specified, the output files will be located in the same folder as the input files.

7.2.2 Sample BIP2 → SysML transformation

One can convert for example the ComplexPackage [12] model from a BIP2 serialized `.xmi` file[4] to a SysML `.xmi` file with the following command:

```
java -jar Bip2ToSysML.jar --mode direct --input-file /some/path/ComplexPackge.xmi \
--output-file /output/path/ComplexPackage_SysML.xmi
```

7.2.3 Sample SysML → BIP2 transformation

The command to convert a SysML `.xmi` model to (a) BIP2 serialized file(s) is similar:

```
java -jar Bip2ToSysML.jar --mode inverse --input-file /some/path/ComplexPackage_SysML.xmi \
--output-file /output/path/
```

7.3 CLI usage additional information

7.3.1 Defining the logging level

In order for the transformation tool to print more debugging information, the logging level can be specified with the argument `--level LOGGING_LEVEL`, where `LOGGING_LEVEL` can be one of the following levels: `SEVERE`, `WARNING`, `INFO`, `CONFIG`, `FINE`, ... (the levels are those specified in the documentation of the `java.util.logging.Level` class [14]).
In order to obtain maximum information during the transformation, append the argument `--level ALL` to the transformation command.

7.3.2 Serializing BIP models

When doing a direct transformation, the BIP input file is the serialized version of the meta-model instance created during the parsing of the `.bip` file. This input BIP file is an `.xmi` file, and it can be obtained with the following arguments appended to the BIP compiler (`bipc.sh`) command: `-s xmi -so serialized`. These arguments specify that the output format for the serialized file is `xmi`, and that the output folder is `serialized/`.

For the "hello-world" example that is on Verimag documentation website [13], this gives :

```
bipc.sh -I . -p HelloPackage -d "HelloCompound()" -s xmi -so serialized
```

The xmi file(s) created in the `serialized/` folder can then be used as input files for a direct transformation.

[4] Generation of BIP serialized models is explained in section 7.3.2

7.3.3 MD Workbench license

You need a valid SODIUS MD Workbench license in order to use the transformation tool. You can specify it either by:

- adding a `mdw.lic` file[5] containing the license information in the `modules/sodius/` folder

```
modules/
└─ sodius/
   └─ mdw.lic The SODIUS license file
```

- specifying the license location as a java *System property* with the following argument: `-Dmdw.license=@license_server` appended to the java command that executes the tool. i.e. `java -Dmdw.license=@license_server -jar BIP2ToSysML.jar --...`

7.4 MagicDraw extensions

There are several files (located in the `ExtendedSysML` folder) that are used for the integration of the BIP2 UML Profile, as described in section 3.2.1:

- `BIP2.mdzip` : this file contains the BIP2 Profile stored in `mdzip`[6] format, and it is needed by MagicDraw when opening a BIP SysML model.
- `descriptors/BIP Diagram descriptor.xml` This file re-defines MagicDraw behavior when it comes to the drawing of an "Internal Block Diagram", as explained in section 3.2.2.

7.4.1 Importing and using the custom profile

We need to inform MagicDraw of the location of the BIP profile before it can display and apply BIP stereotypes to our model. We recommend copying the BIP2Profile.mdzip file that can be found in the `ExtendedSysML` folder of our project into the `profiles` folder folder of the *MagicDraw installation*. Doing so will help you faster select the BIP profile when opening a model coming out of the transformation process.

To use the BIP profile when creating a project from scratch, navigate to `File >Use project >Use local project` and select the BIP2Profile.mdzip file. You should be able to see in the project containment tree that the BIP profile was imported.

7.4.2 Importing and using the custom descriptor

In order to be able to display and use SysML diagrams customized for BIP, you need to import the custom descriptor we prepared for this project. To accomplish this, navigate in the menu the following way: `Diagrams > Customize`. From the dialog window that appeared on the screen, you should be able to import a descriptor. Select the descriptor contained in the "ExtendedSysML/descriptor/" folder and restart MagicDraw.

[5] The license file is a text file complying with the Flexera license file format. For a standard license setup, its content is the following:
`SERVER someServer.epfl.ch 27000`

[6] The mdzip file format corresponds to a zip-compressed MagicDraw project

From now on, you can create BIP-adapted SysML diagrams by pressing the "Create Diagram" button on the top of the screen (Ctrl + N). The customized descriptor should be located in the SysML category.

Tip

When creating a new diagram, MagicDraw will prompt you to indicate which elements you want to initially display. By pressing the Shift key during a click on the containment tree, you can automatically select every sub-element of the selected entity.

7.5 Tool limitations

Because the transformation software is a prototype, the program may fail when encountering "corner cases". Note that the following cases are not supported:

- Cyclic dependencies between BIP packages
- Two levels of nested interaction definitions in BIP connectors
- In a BIP Atom, if an `ExportPortDeclaration` (see section 5.1.3) references more than one `internalPort`, then only the first `internalPort` will be set up as exported in the SysML model

Moreover, invalid BIP or SysML models fed to the transformation program will most likely raise an exception. You may be able to locate the source of the issues based on the output of the Java logger and the exception message. However, due to time constraints we did not implement an extensive validity check on the inputs. Providing as much information as possible to the user in case the transformation could not be completed would be a welcome improvement to the project.

8 Case study 1: Multiple layers nested compound

In this section, we will document how the various parts of a simple BIP compound are converted. The reference compound is the `ComplexCompound` with hierarchical components presented on the BIP documentation website tutorial section [12]. The BIP code of this compound is available in appendix A.1.

8.1 Port type

The BIP language works by connecting components through typed ports that can convey information. The syntax used to define these typed ports is straightforward, as well as their meta-model representation.

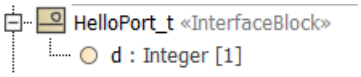
BIP code

```
1 port type HelloPort_t(int d)
```

Meta-model representation

SysML representation

The converted SysML port has a list of typed attributes corresponding to the list of variables defined in the BIP port type, and the ≪InterfaceBlock ≫ stereotype is applied to it.



The port will be represented on SysML internal block diagrams as a ≪ProxyPort≫ , with the following icon: 

8.2 Atom type

A simple BIP Atom is presented here: it has one exported port, and its behavior only defines one state (`LOOP`). There is a transition from the state `LOOP` to itself that is labeled by the port `p`. A guard was put on this transition so that it is allowed only if the variable `active` is equal to `1`.

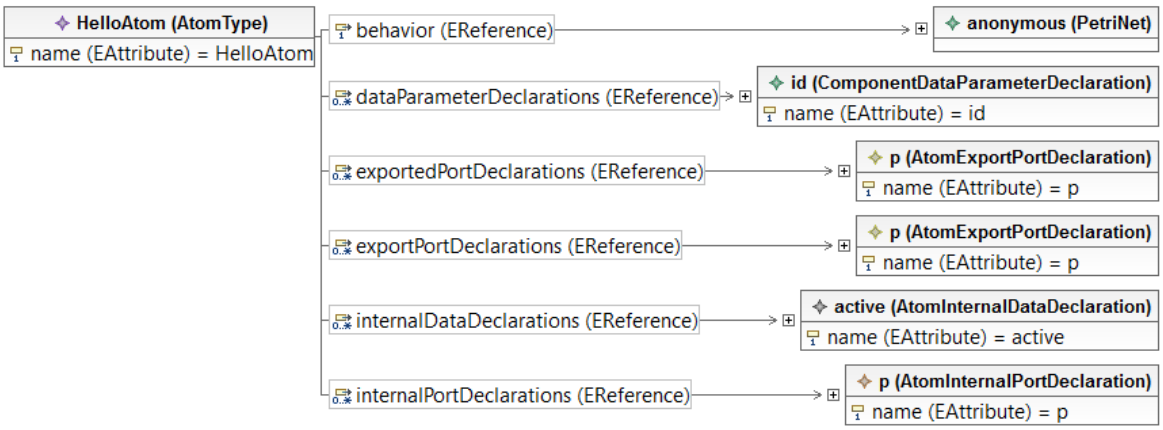
BIP code

```
1  atom type HelloAtom(int id)
2    data int active
3    export port HelloPort_t p(active)
4
5    place LOOP
6
7    initial to LOOP
8      do { active = 1; }
9
10   on p from LOOP to LOOP
```

```
11      provided (active == 1)
12      do { printf("I'm %d, active=%d\n", id, active); }
13  end
```

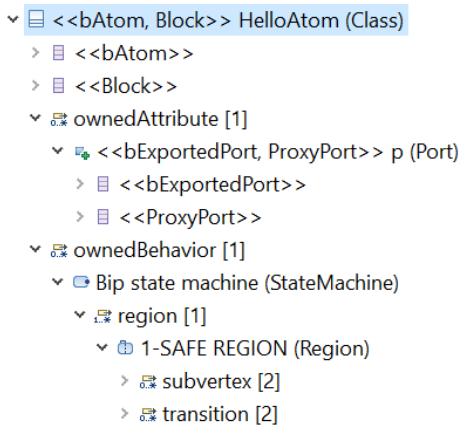
Meta-model representation

The structurally relevant information from the following meta-model representation diagram is that there are `internalPortDeclarations` and `exportPortDelarations`. The former contains is a list of all ports of the Atom, whether `exported` or not, and the latter is a list of references to the `internalPort Declarations`.

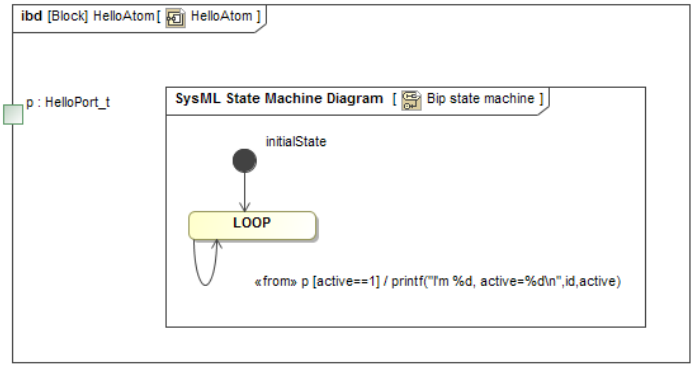


SysML representation

Internal and exported properties are stored in a simpler way in SysML. In addition to applying the ≪bInternalPort≫ and ≪bExportedPort≫ stereotypes, we can set them to be respectively private or public. Private ports will be drawn inside their owning block whereas public ports will be drawn as crossing the boundaries of the block. Note in the following illustration of the SysML model the atom exported port which has a small green "+" sign indicating that it is public and that it will be shown across the block boundaries in diagrams.



Below is the "State Machine" diagram which can be generated from the converted SysML model. Note how MagicDraw displays the labeled port after the keyword "from" (which has nothing to do with stereotypes even if it is surrounded by French guillemets), the guards for transitions between brackets and the executed C function after the slash.



8.3 Connector type

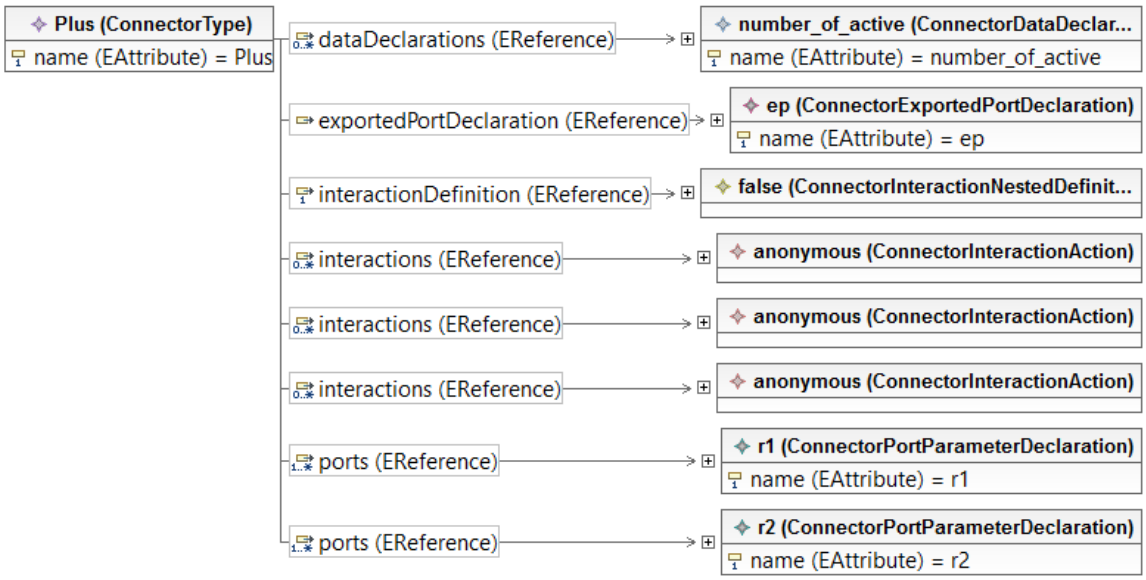
The following BIP connector defines two trigger ports (quotation marks after the port names), r1 and r2, and 3 possible interactions. As explained earlier, the `up` and `down` interaction functionalities of the connector are not converted to SysML.

BIP Code

```
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; }

    on r1
      up { number_of_active = r1.d; }

    on r2
      up { number_of_active = r2.d; }

end
```

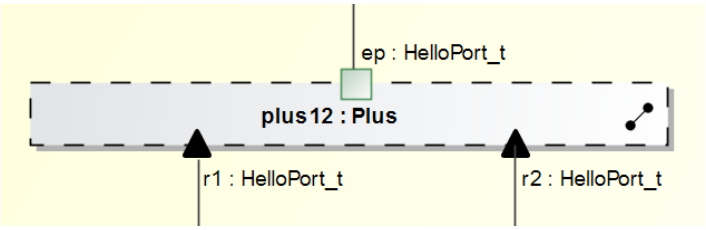
Meta-model representation

The ports are declared individually in the meta-model representation, and the information about their trigger/synchron property is stored in the `interactionDefinition` attribute. The `exportedPort Declaration` attribute declares the optional exported port for the connector. The various interactions are not transformed in SysML (they contain the `up` and `down` expressions, a guard and the list of ports that activate them).



SysML representation

Thanks to the customized descriptor, connector are drawn as a gray blocks in the SysML diagrams. The ports are displayed on the boundaries of the block according to their stereotype: the triangle for ≪bTriggerPort≫ and the circle for ≪bSynchronPort≫ .



8.4 Compound type

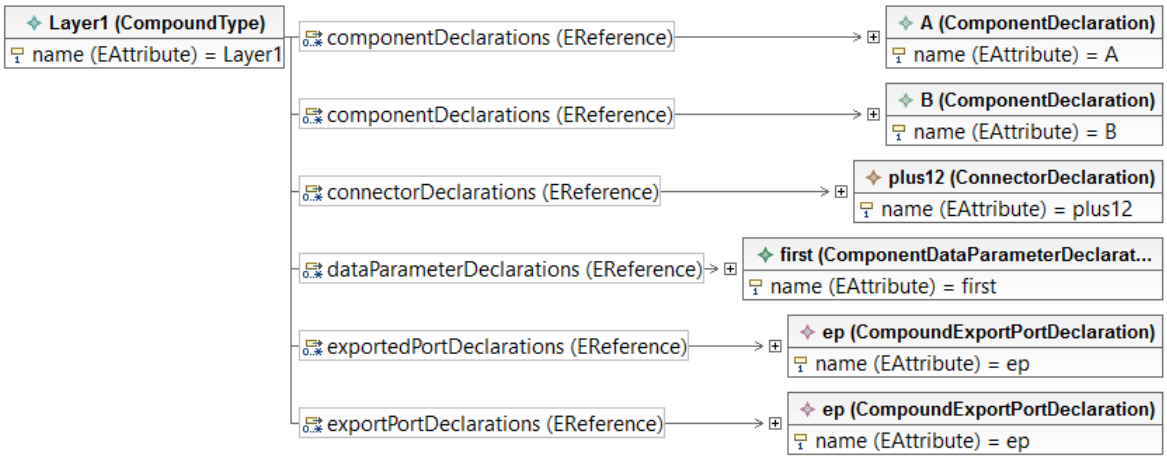
The following compound defines two components, A and B, and connects them using the connector type `Plus` defined earlier. Note how the compound offer external access to the `plus12` connector by exporting the `plus12.ep` port.

BIP Code

```
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
```

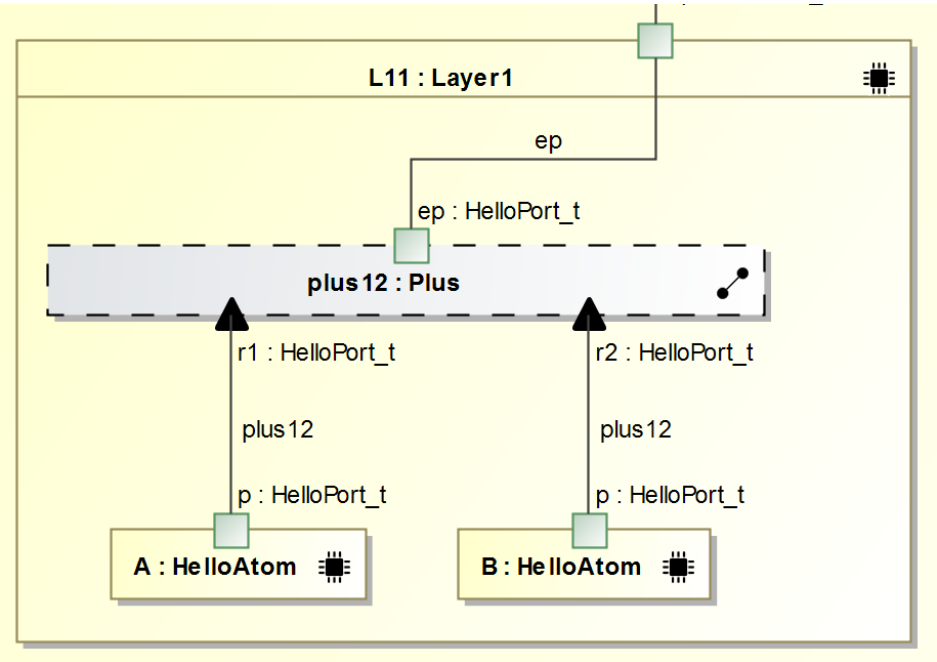
Meta-model representation

Each connector/component of the compound is declared in the list `connectorDeclarations` or `componentDeclarations` of the compound type depending on their nature. See that for this compound, the exported port is declared in the `exportPortDeclarations` list. If we had expanded the view, we would have seen that despite having the same name, it actually references the exported port of the connector `plus12`.



SysML representation

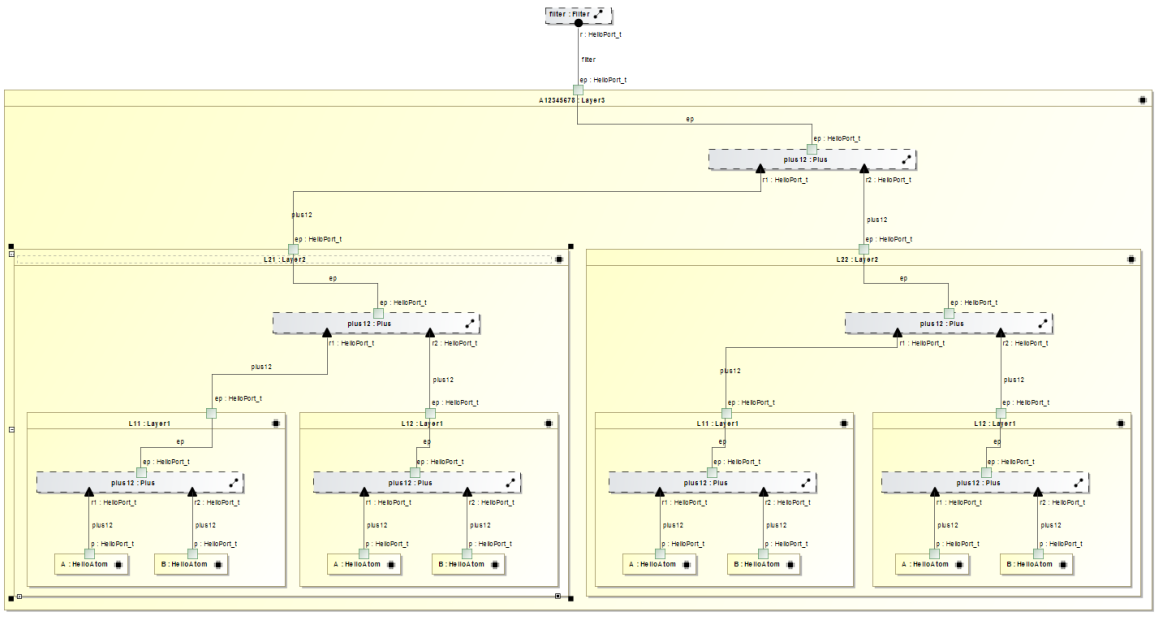
The SysML compound is a ≪Block≫ which has three internal parts. Remember that normally internal parts are typed by the stereotype ≪PartProperty≫ , but due to limitations of MD Workbench we could not apply it to the compound block. Therefore, we left the internal parts as simple UML properties. In SysML, binary connectors are actually children of the compound block.



8.5 Encompassing compound

When the compounds are more complex, some work might be needed before obtaining a graphically pleasing diagram. Fortunately, it was not the case for this model. The following diagram is the one obtained directly after opening the converted SysML model with MagicDraw and creating the BIP diagram described in section 3.2.2.

The nested structure of the encompassing compound is correctly represented, as well as the connections between the various components and connectors:

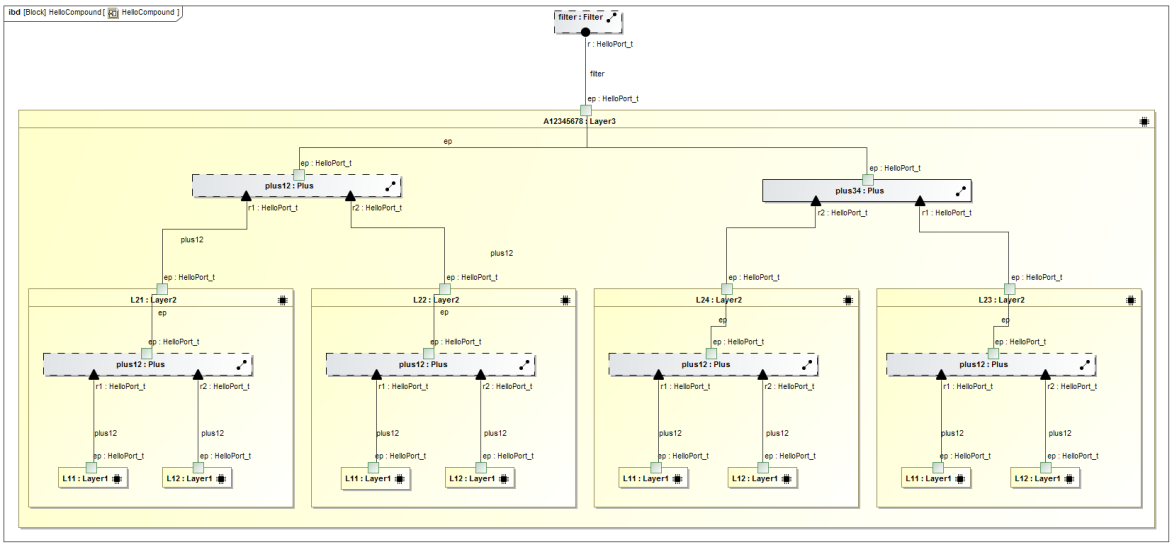


8.6 Modifying the SysML representation and generating the corresponding code

We will now demonstrate how modifications on the SysML representation affect the BIP code. To do this, we will add two new `Layer2` compounds `L23` and `L24` connected by a new `plus34` connector. The following steps have to be accomplished in MagicDraw when editing the diagram of the enclosing compound.

1. From the BIP Diagram palette (bottom left corner of the diagram window), drag&drop two component declarations into the top layer.
2. From the BIP Diagram palette, drag&drop a connector declaration into the top layer.
3. Display ports for both the connector and the component declarations using the "Smart Manipulator.
4. Connect the two trigger ports to the components and the connector exported port to the top layer compound exported port.

For readability purposes we hid the internal structure of this first layer of components:



We see that the applied changes were reflected on the structure of the top layer compound. The left part was the original code and the right part is the output of the BIP compiler when given the SysML to BIP transformation as input:

```
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 Layer3 ()

   component Layer2 L21 ()
   component Layer2 L22 ()
   component Layer2 L23 ()
   component Layer2 L24 ()

   connector Plus plus12 (L21.ep, L22.ep)
   connector Plus plus34 (L23.ep, L24.ep)

   export port plus12.ep, plus34.ep as ep

end
```

Note however, that the integer arguments given to the Layer2 components `L21` and `L22` were lost during the round-trip transformation as they do not directly impact the structure and are thus ignored by our implementation.

9 Case study 2: Housekeeping payload

We propose in this section a second example of conversion but without all transformation details, since the transformation procedure has already been discussed at great lengths and with numerous illustrations. Of course, the transformation process remains strictly identical. In the following figure is illustrated how a BIP model for the housekeeping payload component for the CubETH satellite [1] was illustrated by hand using lucid charts diagrams. We do not show the corresponding BIP code as it is not necessary to understand this transformation.

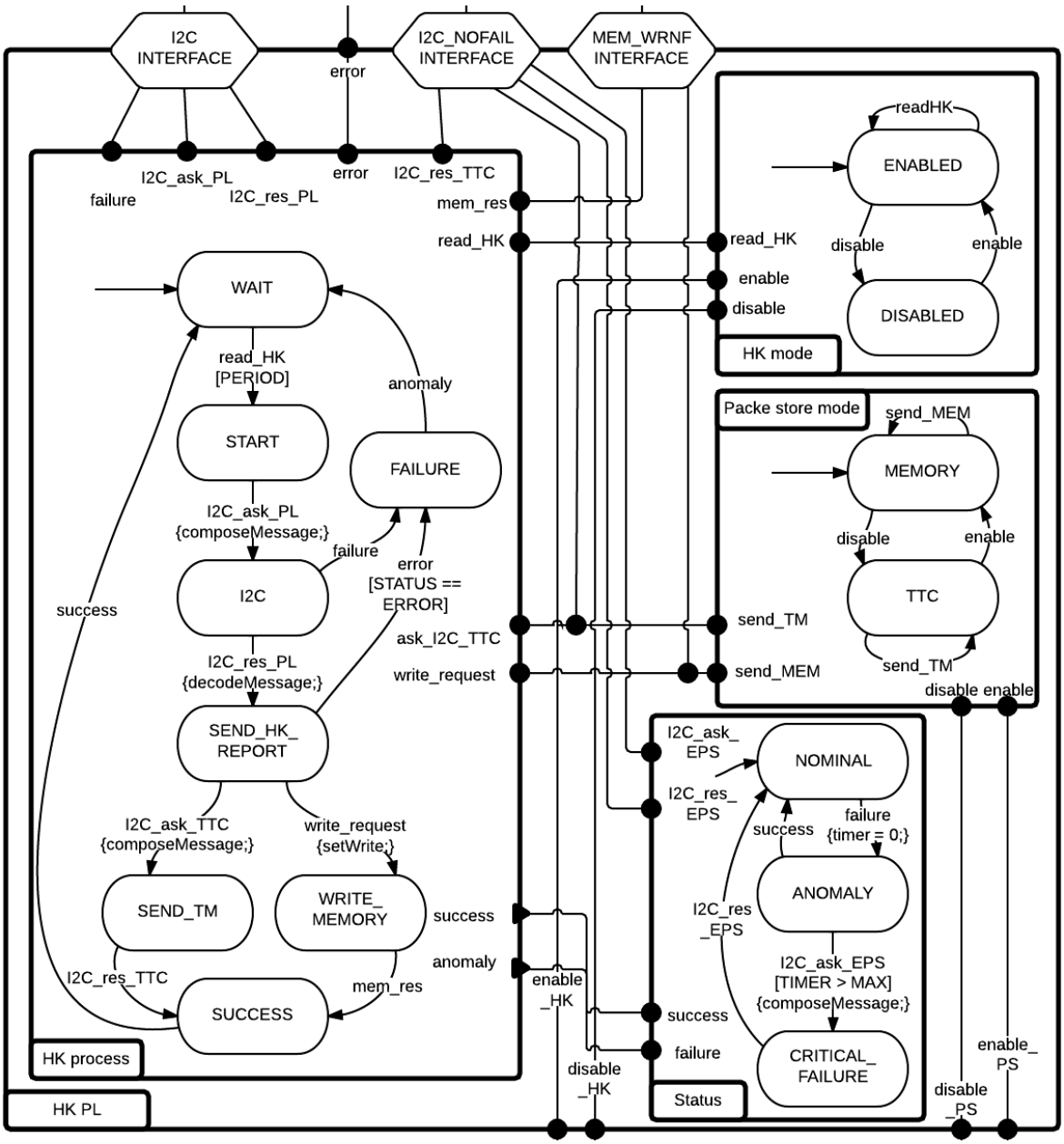


Figure 7: Housekeeping payload compound in BIP illustrated with lucidcharts diagrams by M. Pagnamenta in [2]. Note the positioning and wiring between its four internal components.

The next figure shows the corresponding SysML model that was generated when given the housekeeping payload BIP code as input. Of course, due to the complexity of the payload compound, automatically displaying the generated compound in MagicDraw does not give such a clean result immediately. However, with a bit of reorganization it is quite easy to reproduce the desired representation. Element sizes are rather small since the SysML diagram was not specifically adapted to be printed on a report but the general structure as well as the wiring between components should be recognizable nonetheless.

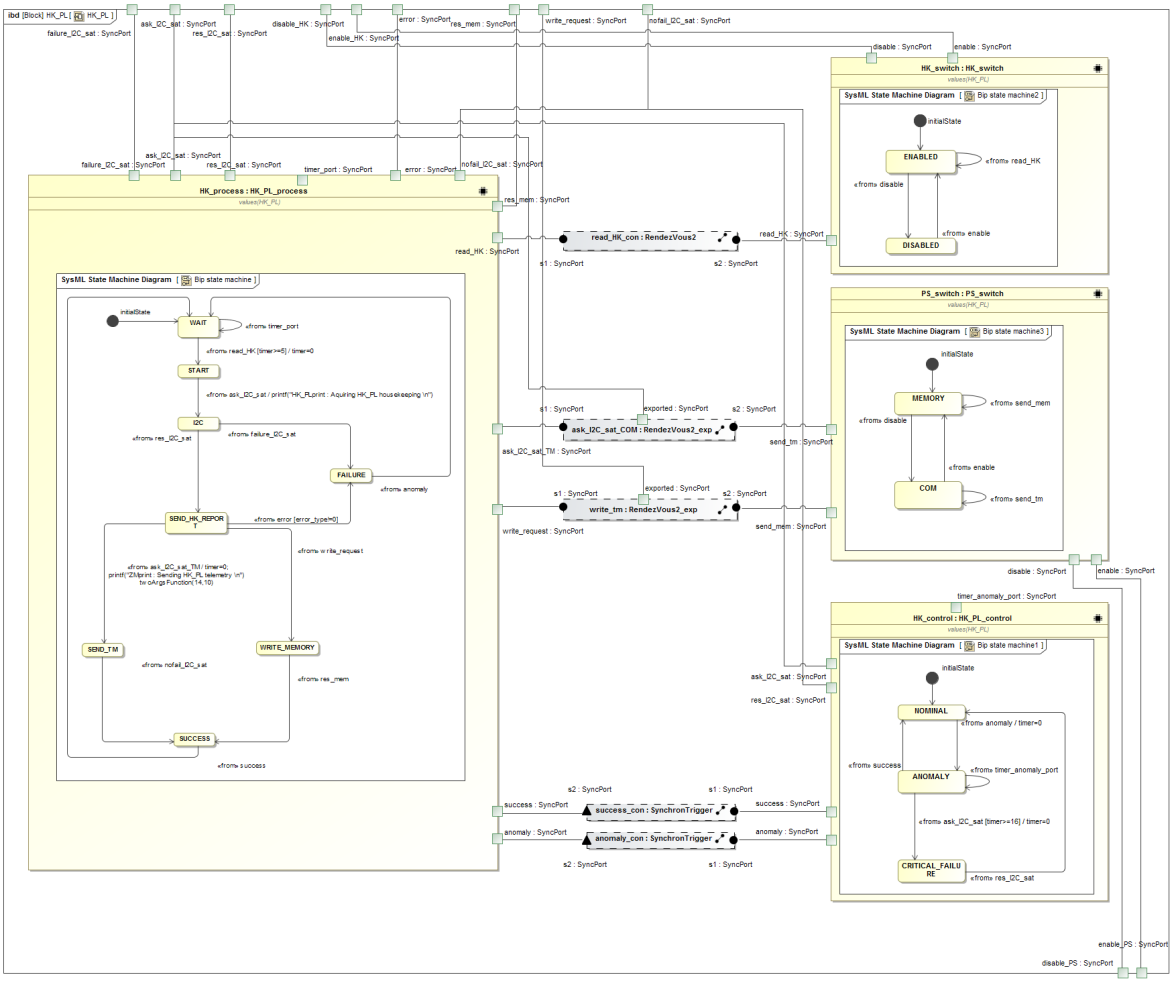


Figure 8: Housekeeping payload compound in BIP illustrated with SysML internal block diagram. Components can be reorganized to generate a representation that is as close as possible from the design idea.

10 Conclusion

10.1 Results

During this project we studied and tested the possibility of expressing BIP models using the SysML profile for UML. We achieved a structural transformation from BIP to SysML using the MD Access and MD Workbench tools conjointly with the Java programming language. MD Access plugins allowed us to read BIP meta-model instances and to create UML models profiled with SysML and the BIP Profile. We used Java to implement the transformation algorithm.

We realized a custom UML profile for expressing BIP-specific elements and customized the standard SysML modeling tool, MagicDraw, to visually emphasize the BIP model elements when opening models generated by our software.

We achieved a structural transformation from SysML to BIP which allows the BIP compiler[7] to regenerate the BIP source code corresponding to the model instance. However, since both forward and backward transformations are purely structural, we lost information during the round-trip transformation. In other words, elements which do not impact the structure of BIP models such as connector interactions and external function calls were not translated to SysML and could therefore not be retrieved during the backward transformation.

Using the inverse transformation it was possible the recreate BIP code "from scratch", by creating SysML models with a graphical tool and applying the correct BIP stereotypes to the model elements, although description of state machines has been, to some extent, tedious to realize. This was mainly due to the guards and expressions needed for the transitions. A more flexible approach would be helpful for this particular aspect.

10.2 Potential project extensions

10.2.1 Extending the transformation beyond the structural features of a BIP model

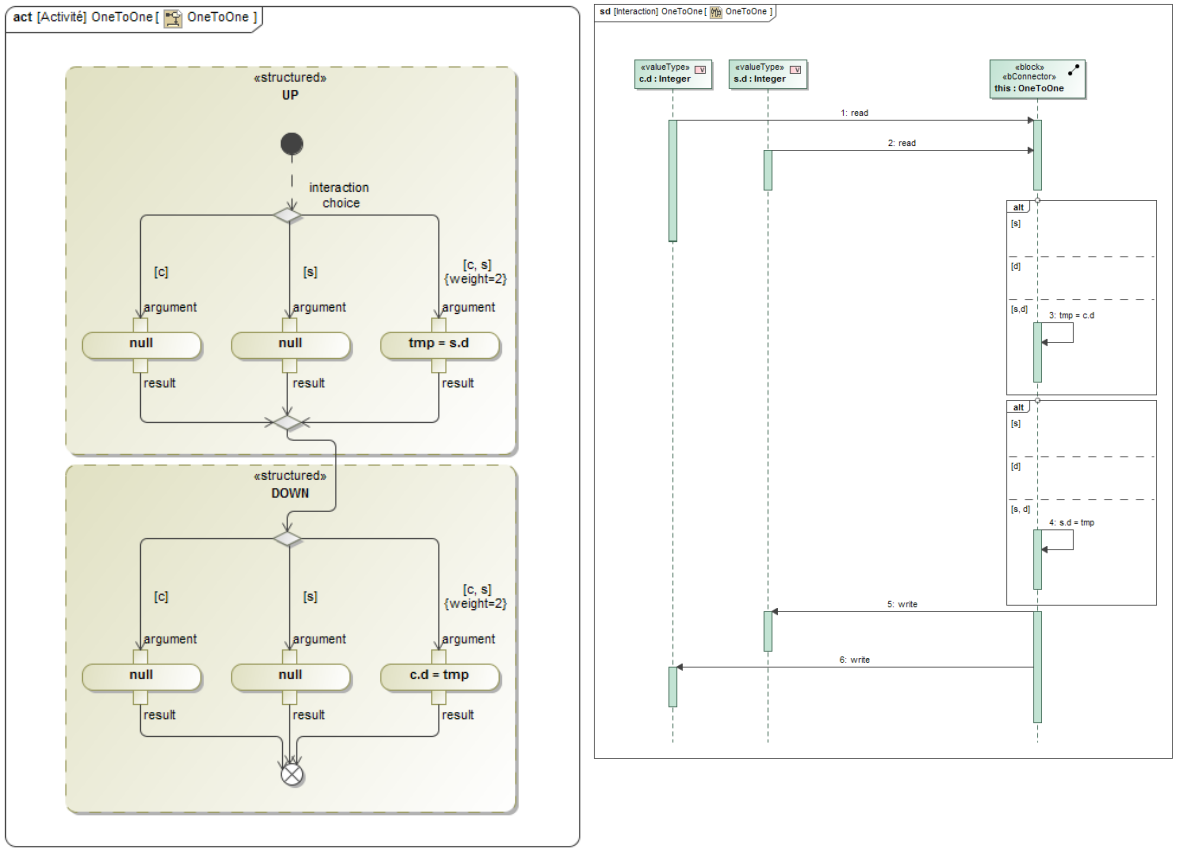
In this project we only transformed BIP models at the structural level. It could be of significant interest to study in more detail how to efficiently store in SysML functional BIP features to allow a lossless round-trip transformation. More specifically, the transformation as currently implemented does not make use, or only partially, of the following elements:

- BIP expressions which are used in guards and transition actions
- Connectors interactions and their assignment expressions which are used for transferring data between components for example.

Although functional properties cannot be directly represented in an internal block diagram, SysML defines more diagram types which are better suited for representing this type of feature. Sequence diagrams can be used to model interactions between components as an exchange of messages and function calls while activity diagrams can be used for modeling flows of inputs and outputs between sub-systems. See figure 9 for an illustration of what an interaction representation could look like using both sequence and activity diagrams. However, thorough research should be made to define what use could be made of such representations.

[7] At the time of writing, the compiler used was a special version enabling generation of BIP code from serialized BIP models [9]. This feature was not proposed with the vanilla BIP compiler.

Figure 9: Possible ways of representing connector interactions; left: activity diagram, right: sequence diagram. The diagram describe a simple variable transfer using a temporary variable during the up operation. Note on the activity diagram the possibility of specifying weight on possible interactions



In a first phase, we could use UML `ValuedExpression` type and extending it if needed with BIP-specific stereotypes to store BIP expressions. We did not have enough time to implement a prototype for that feature but we strongly believe that it is relatively easy to realise. These expressions could then be used for the backward transformation to achieve a round-trip BIP→ SysML→ BIP transformation which preserves all the data contained in the original model.

In a second phase, further research should be made to exactly define the format required by SysML simulations tools such as "Cameo Simulation Toolkit" [11] in order to read and perform computations on the exported BIP expressions.

10.2.2 Developing the transformation as part of the BIP compiler

Use BIP libraries to load BIP xmi files

The sub-package flattening problem (section 4.2) in the BIP MD Access extension prevents the generation of BIP Ecore files that are compatible with the BIP compiler. This problem led us to the realisation that we could have also directly used the BIP libraries to load and work with BIP Ecore model instances. [8]

Take advantage of the BIP compiler "back-end" feature

There are several parts involved in the BIP compiler. The "front-end" part is responsible for parsing `.bip` files and loading the models in memory (i.e., instantiate the BIP meta-model for a given `.bip` file). The "middle-end" part is responsible for operations on the BIP meta-model, and the "back-end" part is responsible for generating the output files corresponding to a BIP model. An example of a BIP back-end is the "cpp" backend, which lets the user generate the C++ code corresponding to a model.

A big improvement for the transformation tool would then be to create a "sysml" backend for BIP, which would simply be added in the BIP compiler `lib/backends/` folder. This would make the generation of BIP SysML models easier: instead of generating BIP serialized models (as explained in section 7.3.2) and transforming them through MD Workbench, one would simply need to append an argument to the `bipc.sh` command.

Use Eclipse libraries to generate the SysML models

Instead of using MD Access UML to write SysML files, it would be interesting to directly use Eclipse libraries to write SysML files. This improvement combined with the two aforementioned improvements would remove the need for the SODIUS proprietary software involved in this project, and it would make generation of BIP SysML models accessible to anyone using the BIP compiler.

10.3 Suggestions for tool improvements

10.3.1 MD Workbench and MD Access

There were several issues that arose when using the MD Workbench software:

- It is (to our knowledge) not possible to specify a custom namespace URI for a resource (the BIP2 uml Profile is an example of such a resource). For this reason, the BIP2 Profile URI specified in the exported BIP SysML models corresponds to the relative path of the `BIP2.profile.uml` file used during the forward transformation. This means that if the file is moved from its original

[8] Parsing xmi file with the BIP compiler is a feature that has been implemented by Jacques Combaz in an extended BIP compiler version[9], and it would be easy to to modify our transformation code to load BIP instance models using these libraries.

location, the BIP2 profile referenced in the `ProfileApplication` section of the exported `xmi` will not be found automatically.

This is why, for example, we need to manually specify the URI of the BIP2 profile when opening a BIP2 SysML model with MagicDraw.

- As mentioned in section 4.2, the sub-packages forming the original BIP2 Ecore meta-model were flattened by the MD Access extension creation tool, with no possibility to keep the complex package structure of the original meta-model.
- We encountered difficulties applying SysML Stereotypes to UML classes with MD Workbench for the following reason: since SysML 1.3 (released in June 2012), the various SysML stereotypes are distributed across 9 sub-packages. However, MD Workbench does not recursively search for Stereotypes when we call the `getApplicableStereotypes()` method on UML classes (as the name suggests, this method is supposed to return a list of applicable stereotypes for an UML object).

  After an email exchange with SODIUS collaborators, it appeared that the aforementioned problem was a known issue and that we had to manually get the stereotypes in the SysML profile. Therefore, at the beginning of the transformation, the program populates a `<String, Stereotypes>` HashMap associating all SysML stereotype qualified names to their stereotype object. We then use the stereotype objects to get and apply SysML stereotypes on UML classes.
- The SysML 1.4 profile is not available in the "MD Access UML" distribution.

10.3.2 BIP ports interfacing

BIP could offer a generic way a regrouping ports into interfaces. This feature was first proposed in Marco Pagnamenta's report [2] for visualizing BIP components in hand-drawn diagrams. This feature can easily be represented in SysML by taking advantage of the ≪ProxyPort≫ stereotype capabilities. Indeed, proxy ports can be typed by ≪InterfaceBlocks≫ and can in turn have nested proxy ports. Therefore, interface blocks can be used to represent groups of BIP ports that are used multiple times in a system. We illustrate in the following figure how proxy ports could be nested to provide an interface corresponding to an AC plug.

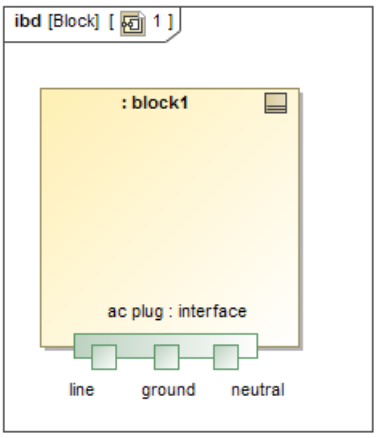


Figure 10: The block contains a port typed by the AC plug interface block. This block contains in turn 3 nested proxy ports for representing the line, neutral and ground pins.

References

[1] infoscience.epfl.ch website: overview of the CubETH nano satellite. *https://infoscience.epfl.ch/record/201520*.

[2] Pagnamenta M. (2014). Rigorous software design for nano and micro satellites using BIP framework [Master's thesis]. Space Center EPFL, École Polytechnique Fédérale de Lausanne.

[3] Verimag website: Rigorous System Design department. [accessed in January 2017]. *http://www-verimag.imag.fr/rsd.html?lang=en*.

[4] BIP Documentation website: Compiler presentation section. [accessed in January 2017]. *http://www-verimag.imag.fr/TOOLS/DCS/bip/doc/latest/html/compiler-engines-presentation.html#the-compiler*

[5] OMG SysML website: SysML language overview. [accessed in January 2017]. *http://www.omgsysml.org/what-is-sysml.htm*.

[6] OMG Consortium website: Object Management Group presentation. [accessed in January 2017]. *http://www.omg.org/gettingstarted/gettingstartedindex.htm*

[7] SODIUS website: MD Workbench software presentation. [accessed in January 2017]. Sodius© *https://sodius.com/en/products/modeldev-tools/mdworkbench*

[8] Verimag website: BIP Compiler download page. [BIP Compiler version used during the project: 2015.04-RC7]. *http://www-verimag.imag.fr/New-BIP-tools.html?lang=en*

[9] Extended BIP compiler version with the ability to read serialized BIP Ecore models. Made accessible by Jacques Combaz in a Private communication (12/2015). *http://www-verimag.imag.fr/ jcombaz/misc/bip-full-2015.12.132108-DEV_x86_64.tar.gz*

[10] NoMagic Inc. website: MagicDraw software presentation. [accessed in January 2017] *https://www.nomagic.com/products/magicdraw.html*

[11] NoMagic Inc. website: Cameo Simulation Toolkit software presentation. [accessed in January 2017] *http://www.nomagic.com/products/magicdraw-addons/cameo-simulation-toolkit.html*

[12] BIP documentation website: Sample multiple layer compound presented in the BIP tutorial. [accessed in January 2017] *http://www-verimag.imag.fr/TOOLS/DCS/bip/doc/latest/html/tutorial.html#hierarchical-components*

[13] BIP documentation website: BIP "hello-world" example model. [accessed in January 2017] *http://www-verimag.imag.fr/TOOLS/DCS/bip/doc/latest/html/tutorial.html#hello-world*

[14] Oracle documentation website. Documentation of the `java.util.logging.Level` class. [Java 8 API] *http://docs.oracle.com/javase/8/docs/api/java/util/logging/Level.html*

The BIP code of this model is available in appendix A.1

A BIP code

A.1 ComplexPackage BIP code

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

  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
```

A.2 Code generated from the DFA SysML model

```

package DFA

    port type numPort (int k)

    connector type WritingHead (numPort tape_side, numPort proc_side)
        define (tape_side proc_side)

    end

    atom type Tape ()

        export port numPort p()

        place READ, END_OF_TAPE
        initial to READ

            on p
                from READ
                to READ

            on p
                from READ
                to END_OF_TAPE

    end

    atom type Processor ()

        export port numPort p()

```

```
        place INIT, REMAINDER_2, REMAINDER_1, REMAINDER_0
        initial to INIT

            on p
                from REMAINDER_2
                to REMAINDER_1

            on p
                from INIT
                to REMAINDER_0

            on p
                from REMAINDER_1
                to REMAINDER_2

            on p
                from REMAINDER_0
                to REMAINDER_1

            on p
                from REMAINDER_0
                to REMAINDER_0

            on p
                from REMAINDER_1
                to REMAINDER_0

            on p
                from REMAINDER_2
                to REMAINDER_2

    end

    compound type EnclosingCompound ()

        component Processor proc_declaration ()
        component Tape tape_declaration ()

        connector WritingHead connector (tape_declaration.p, proc_declaration.p)

    end
end
```