AToMPM User's Manual[*]

Raphaël Mannadiar

(`raphael.mannadiar@mail.mcgill.ca`)

Simon Van Mierlo

(`simon.vanmierlo@uantwerpen.be`)

2016/08/31

[*]This file has version number v0.20. Last revision is dated 2016/08/31.

Contents

1 Client Overview

This section gives an overview of how to interact with the client. Details about modelling, meta-modelling and model transformation activities are given in Sections 2, 3 and 4 respectively.

1.1 Launching the AToMPM Client

Navigate to the URL of an instance of the AToMPM back-end such as `http://atompm.cs.mcgill.ca:8124/atompm` or `localhost:8124/atompm` (if you've installed AToMPM locally).

1.2 The Interface Components

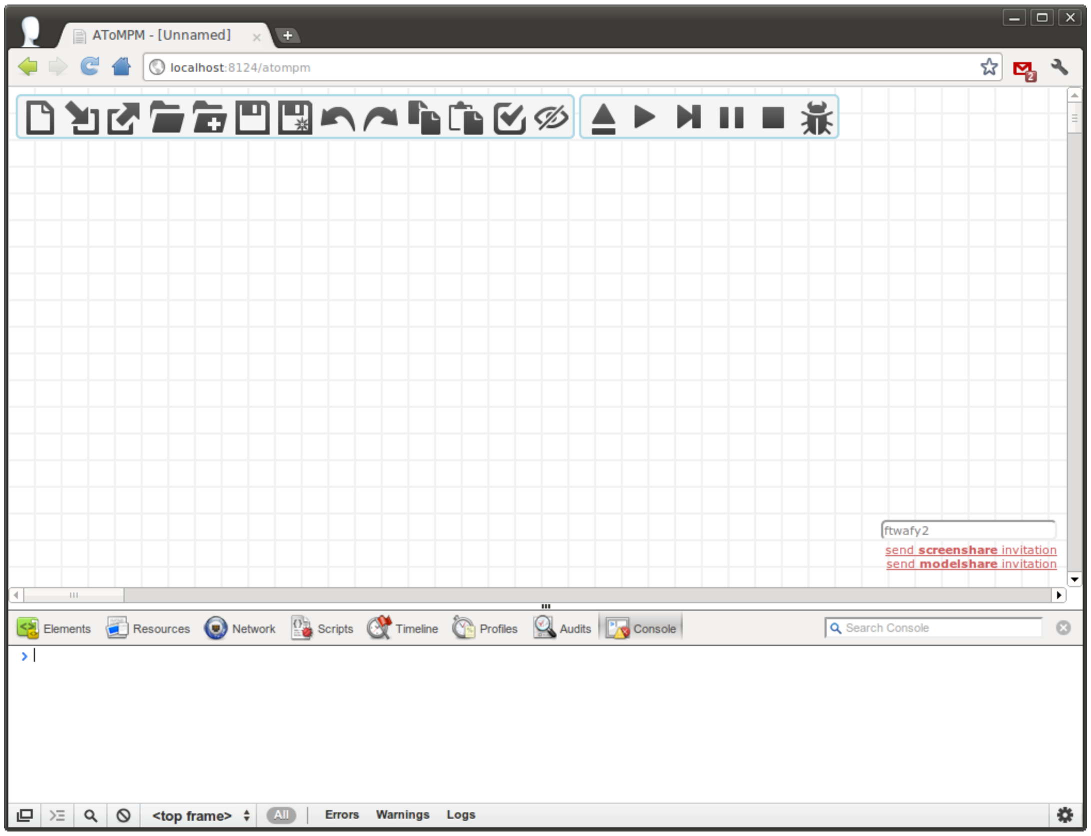


Figure 1.1: The AToMPM client, with the *MainMenu* and *TransformationController* button toolbars loaded.

Figure 1.1 shows the AToMPM client interface. It has 5 components.

The *Dock* (at the top) is where button and formalism toolbars reside. In Figure 1.1, two button toolbars, *MainMenu* and *TransformationController*, are loaded. These are respectively explored in Subsections 1.3 and 4.4.

The *Canvas* (checkered area) is where models are created and edited.

The third and fourth components (at the lower right) are a username field and collaboration links. The former provides access to personal data stored in the cloud. The latter are used to send collaborative modelling invitations.

The final component is the *Console* (at the bottom) which is used to provide the user with various messages and warnings. It can be shown and hidden at will. Means to do so vary from one browser to the next.

1.3 The *MainMenu* Toolbar

#01	Launch a new AToMPM client in a new tab.
#02	(Re-)load a button or formalism toolbar.
#03	Close a loaded toolbar.
#04	Load a model.
#05	Load a model alongside the current model.
#06	Save current model.
#07	Save current model as...
#08	Undo the last performed action.
#09	Redo the last undone action.
#10	Copy.
#11	Paste.
#12	Verify abstract syntax validity constraints (if any) for all loaded formalisms.
#13	Choose formalisms whose entities should be made invisible.

1.4 The *Utilities* Toolbar

The *Utilities* toolbar provides various utility features.

#01	Download an SVG image representation of the current model[1][2].
#02	Load dialog for managing logged-in user's cloud data.

[1]A temporary issue with this is that arrowheads are missing from output.
[2]Note that to make changes in InkScape, you must first Select All and Ungroup.

#03	⬇	Download all of logged-in user's cloud data to local disk.
#04	⚙	Load dialog for customizing logged-in user's personal settings.

1.5 The Canvas

Below is a list of various states the Canvas can be in along with lists of actions available in each state and their corresponding shortcut(s).

When in the **DEFAULT** state,

Action	**Shortcut(s)**
Choose an entity type to create	Left-click on desired type from a loaded formalism toolbar.
Create an entity	Right-click anywhere on the Canvas.
Select an entity	Left-click any entity. This will also select the entity's contents, if any. To select a container without selecting its contents, SHIFT-Left-click it.
Select one or more entity	Left-press anywhere on Canvas, drag selection box around desired entity or entities and release.
Connect entities	Right-press an entity, drag to-be edge to target entity and release.
Edit icon text	SHIFT-Middle-click any text from any icon on the Canvas (this will display a very simple text editor).

When in the **SOMETHING SELECTED** state (i.e., when one or more entity is selected),

Action	**Shortcut(s)**
Unselect selection	Right-/Left-/Middle-click anywhere on the Canvas, or click ESC.
Move selection	Left-press selection, drag preview overlay to desired position and release.
Delete selection	Click DELETE.
Edit first entity in selection	Middle-click selection, or click INSERT, or click COMMAND (this will display the abstract attribute editor).
Enter geometry editing mode	Click CTRL (this will display geometry controls).
Enter edge editing mode	Click SHIFT (this will display editable edge control points).
Snap selection to nearest grid point	Click TAB.

When in the **DRAGGING SELECTION** state (i.e., when right-dragging one or more selected entity),

Action	**Shortcut(s)**
Insert selection into ...	Right-release on top of the target entity.
Un-insert selection from ...	Right-release outside of current container. Containment relationships can also be deleted manually if visible.
Confirm motion	Right-release on the Canvas.
Cancel motion	Click ESC.

When in the **DRAWING EDGE** state (i.e., when dragging to-be edge from source to target entities),

Action	**Shortcut(s)**
Make current line horizontal/vertical	Click TAB.
Create control point	Left-click anywhere, or click CTRL.
Delete last control point	Middle-click anywhere, or click ALT.
Cancel current edge	Left-release anywhere on the Canvas.

When in the **EDGE EDITING** state,

Action	**Shortcut(s)**
Move control point	Left-press any control point, drag it to desired position and release.
Vertically/Horizontally align control point to previous control point	Left-click any control point and click TAB.
Clone control point	Right-click any control point.
Delete control point	Middle-click any control point (extremities and the central control point are not removable).

When in the **GEOMETRY EDITING** state,

Action	**Shortcut(s)**
Scale	Mouse-wheel up/down on scale icon (✥) until preview overlay reaches desired shape.
Scale vertically only	Mouse-wheel up/down on vertical scale icon (⇕) until preview overlay reaches desired shape.
Scale horizontally only	Mouse-wheel up/down on horizontal scale icon (⇹) until preview overlay reaches desired shape.
Rotate	Mouse-wheel up/down on rotation icon (↻) until preview overlay reaches desired shape.
Cancel changes	Right-/Left-/Middle-click anywhere on the Canvas, or click ESC.
Confirm changes	Left-click confirmation (✔) icon.

1.6 Collaboration

AToMPM supports two modes of real-time distributed collaboration, namely, *screenshare* and *modelshare*. In the former, all collaborating developers share the same concrete and abstract syntax. This implies that if one developer moves an entity or cycles to another concrete syntax representation, the change will be replicated for all collaborators. In contrast, in the latter mode, only abstract syntax is shared. This means that all collaborators can have distinct concrete syntax representations and distinct layouts (provided layout and abstract syntax are not intricately related), and are only affected by others' abstract syntax changes (e.g., modifying abstract attribute values).

1.7 Tweaking Default Settings

Several parameters can be tweaked for a more personalized user experience. Their meaning, range of possible values and defaults are detailed below:

Preference Key	**Meaning**
`autosave-delay`	The number of seconds between current model backups, or -1 to disable time-intervalled backups.
`autosave-mode`	When set to *overwrite*, automatic saving overwrites the current model on disk (i.e., has the same effect as if you'd clicked the save button from the MainMenu toolbar). When set to *backup*, automatic saving saves the current model into a temporary file and does *not* overwrite the current model on disk.
`confirm-exit`	When set to *true*, exiting or logging out while the current model contains unsaved changes pops up a warning.
`default-mt-dcl`	The default programming language for all code in model transformation rules.
`autoloaded-toolbars`	Toolbars to load when starting a new AToMPM client.
`autoloaded-model`	Model to load when starting a new AToMPM client.

Preference Key	**Type**	**Default**
`autosave-delay`	integer	15
`autosave-mode`	ENUM(overwrite, backup)	backup
`confirm-exit`	boolean	*true*
`default-mt-dcl`	ENUM(JAVASCRIPT, PYTHON)	PYTHON
`autoloaded-toolbars`	list of toolbar paths	[]
`autoloaded-model`	model path	”

2 Modelling

To create a new model, load one or more formalism toolbars (via button #2 from Subsection 1.3), and create, edit [and connect] entities (via the actions outlined in Subsection 1.5). You can also load an existing model (via buttons #4-5 from Subsection 1.3). Note that you do not need to pre-load formalisms to load models: all required formalisms are loaded automatically.

2.1 Creating Button Toolbar Models

Button toolbar models are a special kind of model in the sense that they can be loaded as models *and* as toolbars. Example button models are provided under `/Toolbars/`. Figure 2.1 shows the model that defines the MainMenu button toolbar. Each entity of the `Button` type has three attributes: `name`, `tooltip` and `code`. `name` is displayed in concrete syntax and is used to resolve the file name of the button's associated toolbar icon: given `name` *n*, there should be an icon named *n.icon.png* in the same folder as the model. `tooltip` becomes the icon's tooltip, i.e., hovering the mouse cursor over a button's toolbar icon displays the value of its `tooltip` attribute. `code` is a code snippet that is evaluated when the toolbar icon is clicked. This code must be valid JavaScript and can make use of the AToMPM Client API, detailed in the following Subsection. Finally, the layout of `Button` entities dictates their order in the toolbar: higher and leftmost buttons are shown first.

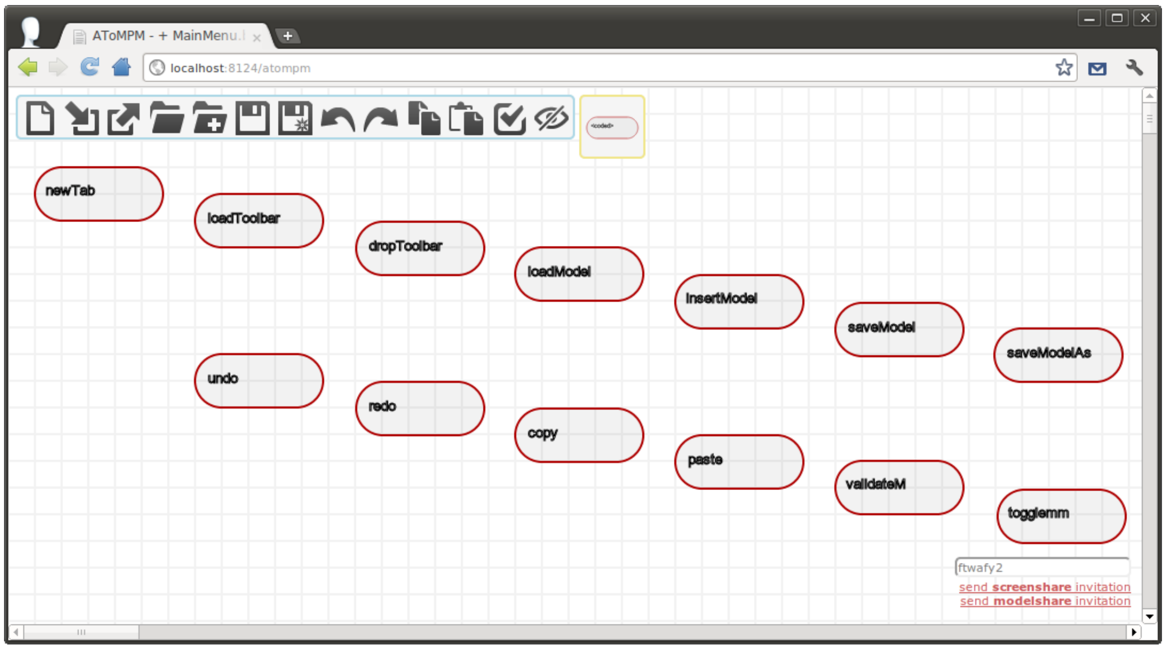


Figure 2.1: The model that defines the MainMenu button toolbar.

2.2 Client API

The following functions can be accessed from within the `code` attribute of `Button` entities in button toolbar models and from the Console. Note that several button toolbar models which make exhaustive use of these functions are provided under `/Toolbars/`.

```
function _compileToASMM(fname)
```

Compile a model, specified by its full path (relative to the current user's trunk), into an abstract syntax meta-model. An error message will be produced if the specified model is not a model of a formalism's abstract syntax. As of now, such models may only be expressed in the bundled `SimpleClassDiagram` or `EntityRelationship` formalisms. Also, note that a recommended (but not enforced) naming convention is that models of formalism abstract syntax be named "*FormalismName*MM.model".

`function _compileToCSMM(fname)`

Compile a model, specified by its full path (relative to the current user's trunk), into a concrete syntax (or icon definition) meta-model. An error message will be produced if the specified model is not a model of a formalism's concrete syntax. As of now, such models may only be expressed in the bundled `ConcreteSyntax` formalism. Note that enforced naming conventions are that models of formalism concrete syntax must be named "*FormalismName*.*description*Icons.model", and that exactly one concrete syntax definition should be named "*FormalismName*.defaultIcons.model".

`function _compileToPatternMM(fname)`

Produce pattern abstract and concrete syntax meta-models[3] given an abstract syntax meta-model, specified by its full path (relative to the current user's trunk). An error message will be produced if the specified meta-model is not an abstract syntax meta-model.

`function _copy()`

Copy selected entities (if any). Note that it is impossible to copy edges if any of their extremities aren't also selected.

`function _exportSVG(fname)`

Prompt the user to download an SVG image representation of the current model and save it under the given filename. When `fname` is omitted, target filename defaults to "model.svg".

`function _getUserPreferences(callback[,subset])`

Call `callback` with the contents of the logged in user's preferences file (see Section 1.7), or a subset of it. When `subset` is defined, it should be an array of desired user's preference keys.

`function _httpReq(method,url,params,onresponse[,sync])`

Perform a synchronous or asynchronous HTTP request given an HTTP method (GET, PUT, POST or DELETE), a URL, a key-value dictionary of parameters, and a function that should handle the request's response. The `onresponse` function should expect 2 parameters: `statusCode` and `responseData`.

[3]Pattern meta-models are discussed in Section 4.

function _insertModel(fname)

Load a model, specified by its full path (relative to the current user's trunk), alongside the current model.

function _loadModel(fname)

Load a model, specified by its full path (relative to the current user's trunk), overwriting the current model, if any.

function _loadToolbar(fname)

(Re-)Load one or more button or formalism toolbars, specified by their full path (relative to the current user's trunk).

function _openDialog(type,args,callback)

Pop-up a dialog box for user interaction. Valid values for the `type` parameter are `_CLOUD_DATA_MANAGER`, `_DICTIONARY_EDITOR`, `_ENTITY_EDITOR`, `_ERROR`, `_FILE_BROWSER`, `_LEGAL_CONNECTIONS`, `_LOADED_TOOLBARS` and `_CUSTOM`. Each type requires its own set of arguments encoded in the `args` parameter. As for the `callback` parameter, it is a one-argument function called with the sum of the user's input when and if the user "OK"es the dialog.

The `_CLOUD_DATA_MANAGER` dialog is used to manage user data stored within the user's personal cloud space. Its `args` parameter should be a key-value dictionary with the following optional keys: "extensions", "read-only" and "title". Values for the "extensions" key should be arrays of regular expressions describing allowed file names. When omitted, all file names are allowed. Values for the "readonly" key should be booleans denoting whether or not the provided dialog should allow cloud data modifications. When omitted, modifications are allowed. Finally, the "title" argument is a string that denotes the dialog's title.

The `_DICTIONARY_EDITOR` dialog is used to read and edit arbitrary *typed* dictionaries[4]. Its `args` parameter should be a key-value dictionary of the form:

```
{
    data           : ``< a typed dictionary >'',
    ignoreKey       : ``< a function that takes 2 parameters, a key and a value, and
                          returns true if they should be shown in the editing dialog >'',
    keepEverything : ``< a function that returns false if only updated key-value pairs
                          should be remembered by the editing dialog >'',
    title           : ``< an optional dialog's title >
}
```

The `_ENTITY_EDITOR` dialog is used to read and edit abstract entity attributes. Its `args` parameter should be a key-value dictionary of the form:

```
{
    uri  :  ``< some entity URI >''
}
```

[4]See Section ?? for an example of a *typed* dictionary.

The _ERROR dialog is used to report an error (e.g., attempting to connect elements that can not be connected) to the user. Its `args` parameter should be a string of text describing the error.

The _FILE_BROWSER dialog is used to browse and select files stored within the user's personal cloud space. Its `args` parameter should be a key-value dictionary with the following optional keys: "extensions", "multipleChoice", "manualInput" and "title". Values for the "extensions" key should be arrays of regular expressions describing allowed file names. When omitted, all file names are allowed. Values for the "multipleChoice" key should be booleans denoting whether or not several files can be selected simultaneously. When omitted, only one file can be selected at a time. Values for the "manualInput" key should be booleans denoting whether or not manual file name entry should be permitted. When omitted, manual file name entry is disabled. Finally, the "title" argument is a string that denotes the dialog's title.

The _LEGAL_CONNECTIONS dialog is used to provide the user with a choice of legal connection types between entities he/she is trying to connect when more than one such connection type is available. Its `args` parameter should be a key-value dictionary of the form:

```
{
    uri1          :  ``< source entity URI >'',
    uri2          :  ``< target entity URI >'',
    ctype         :  ``containment'' | ``visual'',
    forceCallback : true | false
}
```

The `forceCallback` argument indicates whether or not the dialog callback function should be called in the event where no legal connections are available. When unset, the default behaviour is to pop up an error.

The _LOADED_TOOLBARS dialog is used to select loaded button and formalism toolbars. Its `args` parameter should be a key-value dictionary with the following keys: "multipleChoice", "type" and "title". Values for the "multipleChoice" key should be booleans denoting whether or not several toolbars can be selected simultaneously. Values for the "type" key should be ``` ``metamodels'' ```, ``` ``buttons'' ``` or `undefined`. Finally, the optional "title" argument is a string that denotes the dialog's title.

Last but not least, the _CUSTOM dialog enables entirely user-specified dialogs. Its `args` parameter should be a key-value dictionary of the form:

```
{
    widgets  :  [ < ..., widgetDescription_i, ... > ]
    title    : ``< an optional dialog's title >
}
```

where *widgetDescription_is* have the form

```
{
    id       :  ``< widgetId >'',
    type     :  ``input'',
    label    :  ``< input label >'',
    default  :  ``< default value in input field >''
}
```

for input fields, and

```
{
    id              :  ``< widgetId >'',
    type            :  ``select'',
```

```
        choices         :  [ < ..., ‘‘< choice_i >’’, ... > ],
        multipleChoice  :  true | false
    }
```

for lists of choices.

function _paste()

Paste copied entities (if any). Note that copied entities may originate from another AToMPM client.

function _redo()

Redo the last undone action.

function _saveModel([fname,backup])

Persist a model, specified by its full path (relative to the current user's trunk), to the user's personal cloud space. If the `backup` flag is set, the provided filename will not be set as the current filename, and the window title will not be altered to indicate that changes have been saved.

function _setInvisibleMetamodels(mms)

Make all entities from the given formalisms, specified via their full paths (relative to the current user's trunk), invisible.

function _setUserPreferences(prefs[,callback])

Update the logged in user's preferences file (see Section 1.7). `prefs` should be a key-value dictionary. Note that `prefs` need not contain keys and values for all existing user preference keys: missing keys will retain their current value.

function _setTypeToCreate(fulltype)

Set the type of entities that will be created when the user creates new entities on the Canvas.

function _spawnClient(fname,callbackURL)

Spawn a new instance of AToMPM. If a model is specified via the `fname` parameter, it is loaded into the new instance. If a callback url is specified via the `callbackURL` parameter, critical information about the new instance is POSTed to it upon its creation.

function _spawnHeadlessClient(context,onready,onchlog)

(To be completed) .

`function _undo()`

Undo the last performed action.

`function _unloadToolbar(tb)`

Close one or more of the loaded button and formalism toolbars, specified via their full paths (relative to the current user's trunk).

`function _validate()`

Verify abstract syntax validity constraints (if any) for all loaded formalisms.

2.3 Remote API

AToMPM also supports a limited "Remote API" that can be used to edit and animate models remotely, e.g., from third-party or synthesized applications. This is achieved by forwarding specially formatted HTTP queries targeted at the back-end to the client. The said queries must have the form:

```
method  :  ''PUT''
url     :  ''< backendURL >/GET/console?wid=< aswid >''
data    :  {''text'': ''CLIENT_BDAPI :: < func >''}
```

where *aswid* is an identifier for the client's associated back-end abstract syntax thread (retrievable by typing `__aswid` in the client Console), and *func* is a string representation of a key-value dictionary of the form:

```
{
    func    :  ''< Remote API function name >'',
    args  :  { < ..., < arg_i : value_i >, ... > }
}
```

The methods accessible via the Remote API are detailed below.

`function _highlight(args)`

```
  args =
    {
        asid                        :  ''< abstract syntax entity identifier >'',
        followCrossFormalismLinks   :  undefined | ''*'' | ''DOWN'' | ''UP'',
        timeout                     :  undefined | < timeout >
    }
```

Highlight the given entity, specified via its abstract syntax identifier, and un-highlight any highlighted nodes. The `followCrossFormalismLinks` parameter indicates whether or not (and which) neighbours along cross-formalism links should also be highlighted. The `timeout` parameter, if specified, indicates the duration of the highlight (in milliseconds).

function _loadModelInNewWindow(args)

```
args =
  {
      fname         :  ``< model file name >'',
      callback-url  :  ``< callback URL >''
  }
```

A Remote API wrapper around the Client API `_spawnClient()` function.

function _tag(args)

```
args =
  {
      asid     :  ``< abstract syntax entity identifier >'',
      text     :  ``< text to display >'',
      style    :  { < ..., ``< key_i : value_i >'', ...> },
      append   :  true | false,
      timeout  :  undefined | < timeout >

  }
```

Tag the given entity, specified via its abstract syntax identifier, with appropriately styled text *(To be completed)* , appending or overwriting existing tags. The `timeout` parameter, if specified, indicates how long the tag should be displayed (in milliseconds).

function _updateAttr(args)

```
args =
  {
      asid       :  ``< abstract syntax entity identifier >'',
      attr       :  ``< abstract attribute name >'',
      val        :  < new abstract attribute value >,
      highlight  :  true | false
  }
```

Update an attribute of the given entity, specified via its abstract syntax identifier, possibly briefly highlighting the entity to draw attention to the change.

3 Specifying and Compiling Formalism Syntax Models

Defining new formalisms in AToMPM is a three step process. First, one must define and compile a model of the new formalism's abstract syntax. Then, one must define and compile at least one model of the new formalism's concrete syntax. Finally, one should define the new formalism's semantics, or its meaning. The first two steps are explored in this section while the third is explored in the next[5].

3.1 Defining Abstract Syntax

AToMPM currently supports (and is bundled with) two meta-meta-models: `EntityRelationship` and `SimpleClassDiagram`. Instance models expressed in these formalisms can be compiled (via button #2 from Subsection 3.4) into abstract syntax meta-models.

How to define a formalism's abstract syntax in terms of a [connected] collection of `Class` or `Entity` instances is beyond the scope of this manual. Refer to the meta-modelling literature or to the many abstract syntax models bundled with AToMPM. AToMPM's only particularity is the API it supports for meta-modelling constraints and actions, which is detailed in Subsection 3.3.

Note that, when saving models of formalism abstract syntax, a recommended (but not enforced) naming convention is that they be named "$FormalismName$MM.model".

3.1.1 Types

(To be completed)

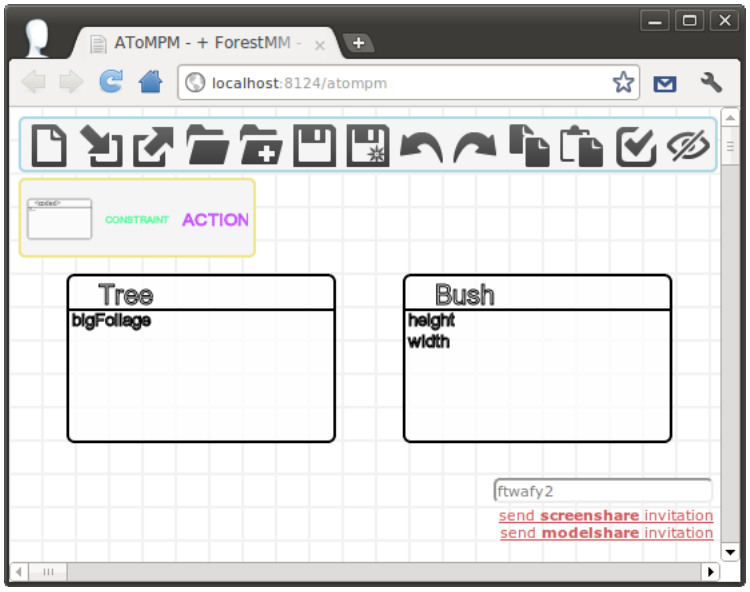
3.2 Defining Concrete Syntax

An abstract syntax meta-model is not sufficient to load a formalism toolbar. Concrete syntax (or icons) must first be associated to the types introduced in the said meta-model. In AToMPM, this is accomplished by defining an instance model of the `ConcreteSyntax` formalism and compiling it (via button #1 from Subsection 3.4) into a concrete syntax (or icon definition) meta-model that can in turn be loaded as a formalism toolbar.

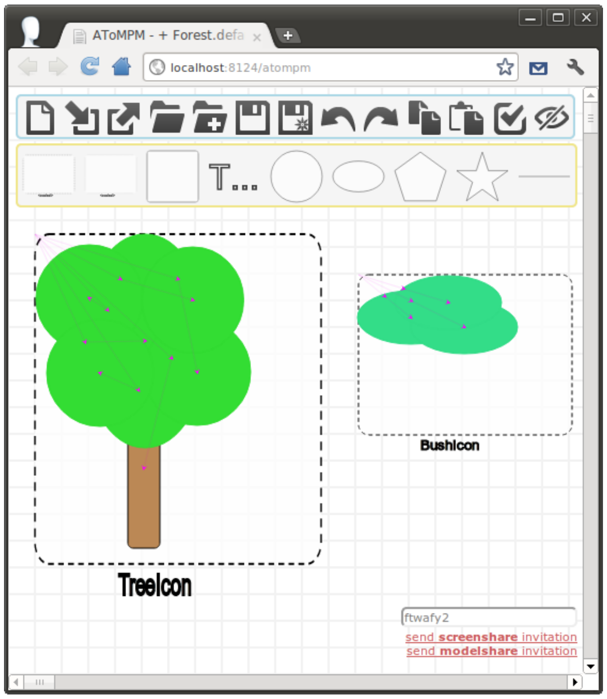
In a formalism's concrete syntax model, each connector type C (from the to-be formalism) must have an associated `ConcreteSyntax.Link` instance with its `typename` attribute set to "CLink". This `ConcreteSyntax.Link` describes the style *(To be completed)* of edges of its associated type and can define an icon (as a combination of styled *(To be completed)* primitive shapes) that will be placed at the center of the said edges. Furthermore, non-connector types T (from the to-be formalism) may or may not have an associated `ConcreteSyntax.Icon` instance with its `typename` attribute set to CIcon. Figure 3.1 gives a very simple example of an abstract syntax model and two associated concrete syntax (or icon definition) models. Numerous more complex examples are bundled with AToMPM.

The combinations of shapes that make up edge center-pieces or entity icons can be further parametrized via `ConcreteSyntax.Icon/Link` and shape `mapper` and `parser` code attributes. The former enable icons and their contained shapes to be altered in response to changes to abstract syntax attribute values. The latter enable changes to icons and/or their contained shapes to effect changes to abstract syntax attribute values. The code stored within the `mapper` and `parser` attributes must be valid JavaScript and can make use of the AToMPM Meta-modelling API, detailed in the following Subsection.

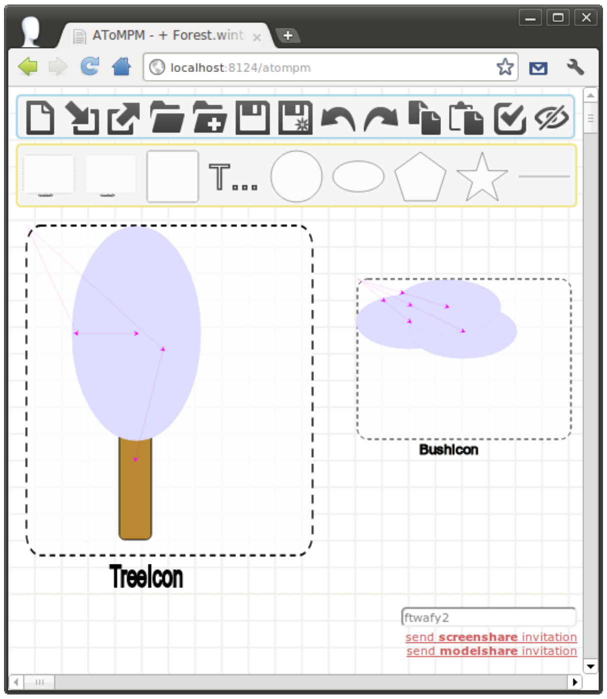
[5]Note that formalism semantics can technically be defined without model transformations (e.g., via elaborate code residing within toolbar buttons) but this is often discouraged by the Model-Driven Engineering community.



(a)



(b)



(c)

Figure 3.1: (a) A very simple abstract syntax model and (b-c) two associated concrete syntax (or icon definition) models.

Note that, when saving models of formalism concrete syntax, two naming conventions are enforced. First, they must be named "*FormalismName.description*Icons.model". Second, exactly one of them must be named "*FormalismName*.defaultIcons.model".

3.3 Meta-Modelling API

The following functions can be accessed from within the `mapper` and `parser` attributes of `ConcreteSyntax` entities, as well as from within abstract syntax constraints and actions (specified within `EntityRelationship` or `SimpleClassDiagram` entities).

For functions with `_id` parameters, if the `_id` parameter is omitted, the abstract syntax identifier of the entity that "owns" the current constraint, action, mapper, or parser, if applicable, is used. Note that in the case of abstract syntax validity constraints, there is no "owning" entity and an appropriate value must always be specified for the `_id` parameter. If an invalid `_id` parameter is ever specified, the current constraint, action, mapper, or parser is immediately interrupted and an error is presented to the user.

`function getAttr(_attr[,_id])`

Return the value of the given attribute from the given entity, specified via its abstract syntax identifier. If no such attribute exists, the current constraint, action, mapper, or parser is immediately interrupted and an error is presented to the user.

`function getAttrNames(_id)`

Return all attribute names of the given entity, specified via its abstract syntax identifier.

`function getAllNodes([_fulltypes])`

Return the abstract syntax identifiers of all entities whose types are contained within the `_fulltypes` array. If it is omitted, return the abstract syntax identifiers of all entities. The notion of *full types* is best explained by example: the full type of a `SimpleClassDiagram.Class` entity is "/Formalisms/__LanguageSyntax__/SimpleClassDiagram/SimpleClassDiagram/Class".

`function getNeighbors(_dir[,_type,_id])`

Return neighbours of the given entity, specified via its abstract syntax identifier. The `_dir` parameter can take on three values: "<" implies that only inbound neighbours should be returned, ">" implies that only outbound neighbours should be returned, "*" implies that neighbours in either direction should be returned. Finally, the `_type` parameter can be set to indicate that only neighbours of the given *full type* should be returned. The notion of *full types* is best explained by example: the full type of a `SimpleClassDiagram.Class` entity is "/Formalisms/__LanguageSyntax__/SimpleClassDiagram/SimpleClassDiagram/Class".

`function hasAttr(_attr[,_id])`

Return *true* if the given entity, specified via its abstract syntax identifier, has an attribute named `_attr`, *false* otherwise.

`function print(str)`

Print the given string to the console that launched the AToMPM back-end.

`function setAttr(_attr,_val[,_id])`

Update the given attribute of the given entity, specified via its abstract syntax identifier, to the given value. Note that this function is only available from within meta-modelling actions. Also, beware the fact that calls to `setAttr()` are not treated like normal model updates, i.e., they do not trigger pre-editing constraints and post-editing actions.

3.4 The *CompileMenu* Toolbar

#1	Compile current model into a concrete syntax (or icon definition) meta-model).
#2	Compile current model into an abstract syntax meta-model.
#3	Compile an abstract syntax meta-model (and its associated icon definition meta-models) into a pattern meta-model.

4 Specifying and Executing Model Transformations

In AToMPM, model transformations and model transformation rules are ordinary models. The former conform to the `Transformation` meta-model, detailed in Subsection 4.2. The latter conform to the `TransformationRule` meta-model, detailed in Subsection 4.1.

4.1 Specifying Transformation Rule Models

Transformation rules in AToMPM are instance models of the `TransformationRule` formalism. They are composed of one Right-Hand Side (RHS) pattern, one Left-Hand Side (LHS) pattern and zero or more Negative Application Condition (NAC) patterns. The contents of these patterns must conform to *pattern meta-models*. Pattern meta-models are slightly altered versions of normal meta-models. Noteworthy alterations are:

- Entities (and their icons) are augmented with `__pLabel` and `__pMatchSubtypes` attributes; *(To be completed)*
- Abstract entities are made instantiable and provided with default icons;
- All entity attributes other than the newly added attributes mentioned above become coded attributes: their values become matching conditions or rewriting actions *(To be completed)* . They should be valid Python or JavaScript and can make use of the AToMPM Transformation Rule API, detailed in Subsection 4.3.

Note that, when saving models of model transformation rules, an enforced naming convention is that their name begin with "R_".

4.2 Specifying Transformation Models

Transformations in AToMPM are instance models of the `Transformation` formalism. They are used to specify how rules and/or transformations should be sequenced together. They are collections of connected transformation steps, where available steps are `Transformation.Rule`s, `Transformation.Transformation`s, `Transformation.Exhaust`s and `Transformation.ExhaustRandom`s.

`Transformation.Rule` entities "point" to a single rule and represent the execution of that single rule.

`Transformation.Transformation` entities point to a single transformation and represent the execution of all the transformations and/or rules contained within it.

`Transformation.Exhaust` entities point to an arbitrary number of rules and/or transformations and their execution is only considered completed when none of them are applicable, with priority given to the top-most (within the list of pointed-to rules and/or transformations) rule or transformation at each execution step.

`Transformation.ExhaustRandom` entities point to an arbitrary number of rules and/or transformations and their execution is only considered completed when none of them are applicable. Unlike, `Transformation.Exhaust`s, the choice of the next rule or transformation (within the list of pointed-to rules and/or transformations) is non-deterministic.

Transformation steps can be connected to each other via three association types: `OnSuccess`, `OnNotApplicable` and `OnFailure`. The meaning of steps S_1 and S_2 being connected by an `On***` association is that, during transformation execution, the flow of control will pass from S_1 to S_2 if the outcome of S_1 is `***`. For `Transformation.Rule`s, a `Success` outcome means the rule was successfully applied, a `Failure` outcome means an error occurred during rule application, and a `NotApplicable` outcome means the rule precondition could not be satisfied. For the three other step types, a `NotApplicable` outcome means that all contained/pointed-to rules and/or transformations completed with outcome `NotApplicable`, a `Failure` outcome means that the last executed contained/pointed-to rule or transformation completed with outcome `Failure`, and a `Success` outcome is produced otherwise.

Note that, when saving models of model transformations, an enforced naming convention is that their name begin with "T_".

4.3 Transformation Rule API

The following functions can be accessed from within LHS and NAC condition code, from within RHS action code, and from within attribute condition and action code. Note that this code can be specified in either Python or JavaScript[6]. The default language is specified in your preferences file (see Section 1.7). To use an alternate language, the first line of the relevant body of code should be "`"[<LANGUAGE>]"`" (e.g., "`"[PYTHON]"`"). A final difference is that while JavaScript code needs only evaluate to the desired value (e.g., *true* for a condition), Python code needs to store its return value in a variable named `result` (e.g., `result = true`, as the final line of a condition).

`function getAttr(_attr[,_id])`

`function getAllNodes([_fulltypes])`

`function getNeighbors(_dir[,_type,_id])`

`function hasAttr(_attr[,_id])`

`function print(str)`

`function setAttr(_attr,_val[,_id])`

All of the above functions operate exactly like the functions of the same name presented in Subsection 3.3 with the exception that entities may be specified via their `__pLabel` as well as via their abstract syntax identifier.

`function httpReq(method,host,url,data)`

Perform a synchronous HTTP request given an HTTP method (GET, PUT, POST or DELETE), a URL and a key-value dictionary of parameters. If `host` is undefined, the request is automatically routed to the AToMPM backend. This can be useful to make use of the Remote API from within rule code.

`function isConnectionType(_id)`

Return *true* if the given entity, specified via its abstract syntx identifier or its `__pLabel`, is a connection type, *false* otherwise.

[6]JavaScript support requires additional back-end modules. See the README file located in the source trunk or contact the maintainer of the AToMPM back-end you're using.

`function session_get(_key)`

`function session_put(_key,_val)`

The *Transformation Session* is a sandbox of sorts that enables miscellaneous user data to be easily accessed and stored across several rule and transformation executions. It is only ever cleared when a transformation is (re-)loaded (via button #1 from Subsection 4.4). The above methods respectively enable retrieving and setting/updating a stored value.

`function sys_call(_args)`

Perform a system call on the machine hosting the AToMPM back-end. An example value for the `_args` parameter is [“ls”, “-l”][7].

`function sys_mkdir(_path)`

Create the given directory (or directories).

`function sys_readf(_path)`

Return the contents of the given file.

`function sys_writef(_path,_content[,_append])`

Write `_content` to the given file, overwriting its contents if the `_append` attribute is set to *false*.

4.4 The *TransformationController* Toolbar

The *TransformationController* toolbar should be used to control transformation execution.

#1	⏏	Choose transformation to run.
#2	▶	Run transformation in continuous mode.
#3	⏭	Run a single rule (the next rule to be run given transformation specification).
#2	⏸	Pause transformation (after currently executing rule, if any, completes).
#3	■	Stop transformation.
#3	🐞	Toggle between transformation debugging on and off. *(To be completed)*

[7]Note that this function can be a tremendous security hazard if the AToMPM back-end is launched from a user account with unchecked privileges.

5 Extending AToMPM *(To be completed)*

how to develop plugins
when to use them

until completed, see `/<username>/__Examples__/Toolbars/` and `/plugins`

A Setting up the AToMPM back-end

This section is only useful if you want to install your own copy of the AToMPM back-end locally. This, as opposed to using an online version such as the one hosted by McGill University's Modelling and Simulation Lab at `http://atompm.cs.mcgill.ca:8124/atompm`.

A.1 Installation

For installation instructions, see the README file located in the source trunk.

A.2 Launch

To start the AToMPM back-end, you must launch two processes from the source trunk:

1. `node httpwsd.js`, and
2. `python mt/main.py`.

If you want to expose your instance of the AToMPM back-end beyond the host machine, you must allow network traffic on port 8124.