System documentation

ABI team 33
Guus Bonnema, Stefan Versluys en Jeroen Kleijn

22/04/2015

Abstract

This document contains the system documentation for the xMAS Model Designer. It describes the design and implementation details of the designer, the integration with verification tools, changes to existing code and techniques used for the implementation.

1 Story tree

Stories of the xMAS model designer project can be found in the Agilefant data in the xMAS repository. To use this data you must set up Agilefant on a web server and import xmas/6-transition/08-agilefant/Agilefant_xMAS.zip

The stories are based on three roles:

- A **researcher** develop formal verification methods to test on xMAS models.
- A **designer** create and verify xMAS models with the xMAS model designer toolkit.
- A **maintainer** that maintains the xMAS model designer toolkit project.

Figure 1 is a small copy of how this story tree looks like in the Agilefant tool.

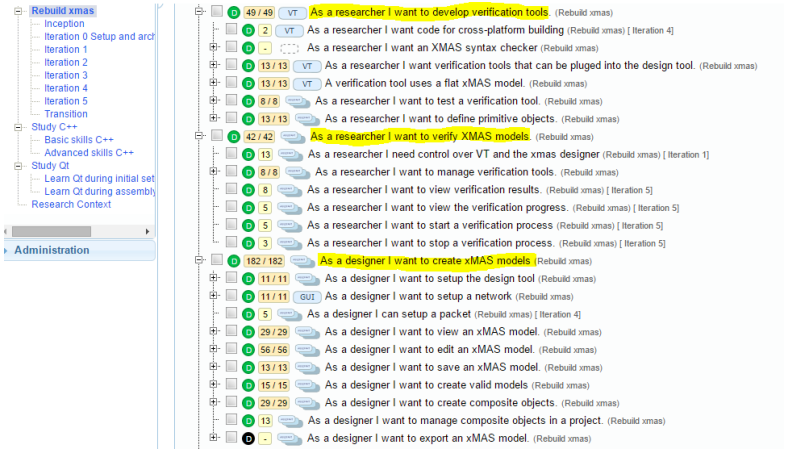

Figure 1: Agilefant story tree

2 Changes to the data model

Due to requirements imposed by the designer, several changes and additions have been made to the data model. Most of these changes are related to the additional support for hierarchical networks in the designer. This section present an overview of new and modified classes and functions.

2.1 XMASNetwork

XMASNetwork is a new class designed to represent an xMAS network. This class can be used to model both flat and hierarchical networks. When used to model hierarchical networks, an XMASNetwork represents a single level or subnetwork in the hierarchy. Multiple networks are combined to form the complete network.

The main responsibility of XMASNetwork is to serve as a container of XMASComponent instances. Internally, XMASNetwork stores the components in a map relating a components name to its in-memory instance. Prior to the introduction of XMASNetwork, the concept of an xMAS network was directly represented by such a map. Verification tools still use this approach, although they could be easily adapted to use XMASNetwork as well.

The 'promotion' of XMASNetwork to its own class definition has two primary reasons:

- The designer uses additional network properties like the canvas size. These properties must be stored somewhere in the data model (i.e. in XMASNetwork).
- Support for hierarchical networks requires management of multiple network models. Using an explicit class to represent networks eases this task.

Extending networks XMASNetwork implements the same extension mechanism as used by the XMASComponent and Port classes. For this, a new class named XMASNetworkExtension has been created. Currently, the designer uses two network extensions. One to store network properties common to all models, and one to store network properties specifically for network models that are to be used as composite objects. In the future, additional network extensions could be defined, for example to store data required to support parameterization of subnetworks.

MemoryPool All components in an XMASNetwork are stored in a MemoryPool for optimal performance. Multiple networks can share a MemoryPool instance. The XMASNetwork constructors optionally take a pointer to a MemoryPool. Passing the same instance to multiple networks will let these networks share the pool. When no MemoryPool pointer (or nullptr) is provided to the constructor, XMASNetwork will create its own MemoryPool instance. In this case, XMASNetwork also takes care of the MemoryPool destruction.

2.2 XMASComposite

When modeling a hierarchical network, composite objects are used to represent subnetworks as black boxes inside a higher-level network. The addition of composite objects to an xMAS network is reflected in the data model through the new XMASComposite class. Like the eight xMAS primitive types, XMASComposite is derived from XMASComponent. As such, XMASComposites can be used in the same way as the primitive components.

When a new XMASComposite object is created, a reference to an XMASNetwork must be passed to the constructor. The composite object will represent an instantiation of this (sub)network.

The sinks and sources of the subnetwork are used as interface ports or gates between the subnetwork and the higher-level network. An xMAS source component will result in an Input port on the composite object. Likewise, each xMAS sink component in the subnetwork leads to an Output port on the composite object.

2.2.1 Hierarchical visitors

Due to the introduction of the new XMASComposite type, the XMASComponentVisitor interface has been extended to support composite objects. Unlike the existing pure virtual visit functions, XMASComponentVisitor provides a default implementation to visit XMASComposites. This way, existing implementations of the interface which weren't designed to support composites aren't affected. The default visit implementation for composite objects does throw an Exception however. So, only flat, composite free networks should be passed to these implementations.

2.3 XMASProject

The xMAS designer application uses the XMASProject class to manage a complete hierarchical model. The root network, available through getRootNetwork(), is the network under construction. For each composite object type used, XMASProject additionally stores its network definition. XMASProject is equipped with member functions to insert new components into the network, remove them from the network and change the name of a component. All of these functions act upon the root network. Adding a new composite object to the model requires that its network definition is added to the project in advance. Function insertComposite() automatically does this if necessary. A subnetwork that is no longer of use can be unloaded using unloadNetwork(). Unloading only succeeds if no composite objects in the project (at any level) depend on the network.

XMASProject is also responsible for the construction and destruction of a MemoryPool. All networks loaded in the project share this instance of the pool.

Note: Currently, the designer assumes that all subnetworks used in a project are stored in the same directory as the root network.

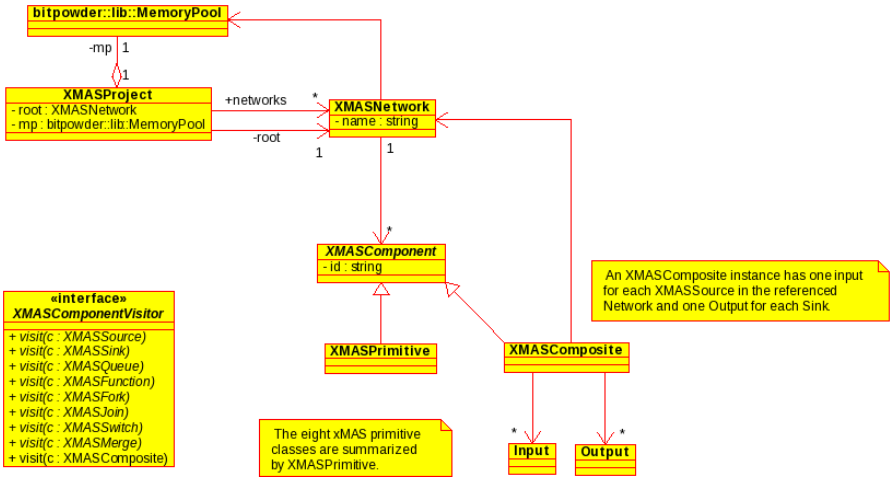

Figure 2: xMAS data-model extended with new classes

2.4 Parser & Exporter

The additional support for hierarchical models and the need to store data used by the designer require a number of modifications to the parser and the exporter.

Position For each component listed in the json data, the parser checks whether canvas data is available in the pos field (see also the file format description). If this is indeed the case, the x, y, orientation and scale fields are read and stored inside an extension of the component (CanvasComponentExtension). If no position data is stored in the json data, the designer will use default values. The exporter has been updated to write the position data, if present, back to json as well.

Composite objects XMASComposite objects are created for all components in the json file with type 'composite'. The parser uses the 'subnetwork' attribute mandatory for composite components to determine what kind of composite object should be created. The parser is not responsible for loading subnetworks. Rather, the caller of the parser must supply a function that is able to map a subnetwork name to an instance of an XMASNetwork. Code that uses the parser should be updated to comply with the new function signature. The implementation in XMASProject can be used as a reference.

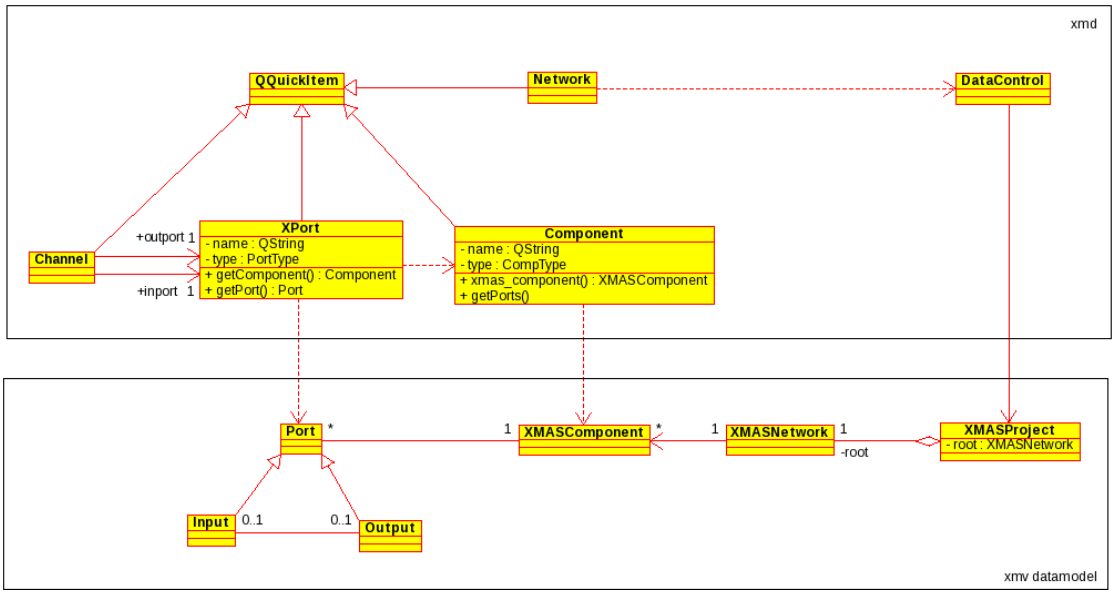

Figure 3: Integration of xmd and xmv datamodel

3 XMD - XMV integration

3.1 QML / C++ integration

3.1.1 Context property

3.1.2 C++ extension class

3.1.3 Interaction xmd / datamodel classes

4 Feedback interface

While analyzing a network, verification tools can generate various types of feedback messages. At this moment, verification tools present this feedback as human readable text messages on the standard output. Although this form is suitable for human consumption when running the verification tools as a standalone application, they are hard to parse and interpret by software like the designer.

Due to time constraints, presentation of feedback from the verification tools has been kept simple. Similar to the standalone application, the text messages are presented to the user in a text console integrated in the designer. Future improvements could enhance the integration between designer and verifications. As an example, problematic components and ports could be highlighted on the design canvas.

To support these kinds of enhancements, a more structured way to pass feedback from verification tools must be used. For this purpose, a feedback message format has been defined along with several functions that can be used by verification tools to generate feedback messages.

4.1 Message format

Feedback messages are structured as follows:

[**sender**] ****type****: **message** (**component list**) (**port list**)

- **sender** is a string to identify the verification tool that sends the feedback message. Each verification tool should define a unique name to identify itself.
- **type** specifies the type of feedback. The following feedback types are defined:
  - **INFO** - informational message, e.g. verification tool has started or finished
  - **WARNING** - non-fatal warning, e.g. non-optimal network configuration detected
  - **FAULT** - fault in the network detected, e.g. a combinatorial cycle
  - **CRASH** - crash / unexpected exception thrown by the verification tool, this indicates a programming error, not a network fault
  - **PROGRESS** - informational message containing progress of a verification tool
- **message** contains the actual feedback message. Verification tools can pass custom text messages. For PROGRESS type feedback, the message field has a fixed format: "progress/total" where both progress and total are positive integers.
- **component list** is an optional list of component IDs to which the feedback applies. The components are listed inside brackets using comma-separated notation and are prefixed by "Comp: ", e.g. Comp: [src1, join2, ...].

- **port list** is an optional list of ports (of components) to which the feedback applies. The ports are listed inside brackets using comma-separated notation and are prefixed by "Port: ", e.g. Port: [src1.o, join2.a, ...].
- messages are terminated by a newline character

4.2 Example

```
[cycle-checker] **INFO**: Starting cycle checker
[cycle-checker] **FAULT**: Combinatorial cycle detected!  Port: [::frk0.a]
[cycle-checker] **INFO**: Finished!
```

4.3 Current state & future work

The "cycle checker" verification tool has been adapted to output its feedback in the described format. Optionally, this tool could be updated to emit progress information. However, progress information is of little value due to the short runtime of this tool.

As mentioned earlier, the designer does not interpret the feedback message structure yet. Extracting the information from the text string can be accomplished through the use of a regular expression. All feedback formatting code is located in the feedback-interface.h/cpp files. Changes to the message format and new code to parse the message string can be added here.

5 XMAS network file format

This document describes the file format used to store XMAS network models. Both the XMAS designer application and the standalone verification tools use this file format. The structure is based on the flat json format used by the checker application provided at the start of this project. Several new fields have been added in order to support hierarchical networks and information specific to the designer. As the new file format is able to support both flat and hierarchical networks, the (recommended) file extension has been changed from fjson to json.

5.1 Root JSON object

Properties:

- "CANVAS" : **CANVAS** object (optional) *NEW*
- "COMPOSITE_NETWORK" : **COMPOSITE_NETWORK** object (optional) *NEW*
- "VARS" : **VARS** object
- "PACKET_TYPE" : **PACKET_TYPE** object
- "NETWORK" : array of **COMPONENT**, describes all components in the network

5.2 CANVAS

NEW Used by the XMAS designer application to define properties of the designer canvas. When no CANVAS information is specified, default values are used.

Properties:

- "width" : width of the canvas in logical units
- "height" : height of the canvas in logical units

5.3 COMPOSITE_NETWORK

NEW Contains designer specific information to use this network as a composite object. If this information is not present, the network cannot be used as a composite object.

Properties:

- "alias" : displayed inside a composite object to denote the objects type (e.g. mesh)
- "image-name" : name of graphical resource used as a symbol to denote the objects type (e.g. mesh.ico)
- "boxed-image" : set to 1 to use the image as a symbol inside a generic composite object (drawn boxed) or set to 0 to use the image as the graphical representation of the entire component, this can be used to draw common macro's like credit counters and delays using the symbols commonly used in literature to denote these macro's.
- "packet" : ???

5.4 VARS

TODO Definition taken from original file format

5.5 PACKET_TYPE

TODO Definition taken from original file format

5.6 COMPONENT

Describes a component in the network.
Properties:

- "id" : **String** - unique identifier
- "type" : **Component_type** enum - type of the component
- "outs" : array of **OUT** - describes all connected output ports
- "fields" : array of **FIELD** *(optional)*
- "pos" : **POSITION** - *new* - position of the component on the canvas

5.7 Component_type

Enumeration:

- source
- sink
- function
- queue
- xfork
- join
- xswitch
- merge
- composite ***NEW***

5.8 COMPONENT [type="source", type="sink"]

Properties:

- "required" : **Boolean 1/0** - set to 1 when this source/sink is an interface port of a composite network that must be connected by the higher level network. set to 0 when this is optional.

5.9 COMPONENT [type="composite"]

NEW

Properties:

- "subnetwork" : **String** - name of the subnetwork
- ("parameters" : **Not determined yet** - reserved property name, to be used in the future to parameterize the network)

Description: The subnetwork reference indicates the relative location of the subnetwork on the filesystem.

E.g. "mesh.json" refers to the network defined in "mesh.json" in the same directory as this network and "spidergon/node.json" refers to the network defined in "node.json" in the subdirectory "spidergon".

Parameterization of composite objects has not been implemented yet. When implemented, composite objects require an additional field in the component description to store the parameters. The property name "parameters" has been reserved for this purpose.

Note: Currently the XMAS designer requires that all used composite networks are defined in the same directory as the root network.

5.10 OUT

Description: Describes channels between the components in the network. For each output port of a component, an **OUT** object describes to which input port of which component it is connected.

Properties:

- "id" : **String** - reference to the target component
- "in_port" : **Number** - index of the input port on the target component

5.11 FIELD

TODO Definition taken from original file format

5.12 POSITION

NEW Stores positional data of the component on the canvas.

Properties:

- "x" : **Number** - x or horizontal position on the canvas
- "y" : **Number** - y or vertical position on the canvas
- "orientation" : **Number** - orientation of the component, measured in degrees clockwise.
- "scale" : **Number** - scale factor of the component