VT - XMD integration

ABI team 33
(Stefan Versluys, Guus Bonnema, Jeroen Kleijn)

20/02/2015

Abstract

* Why did we write this document? * What does it cover? * How to read it?

This document explains why and in what manner the team decided to decouple the designer subsystem XMD and the VT subsystem. The contents documents the background, the problem, the decison and the motivation for the decision, and finally the detailed considerations that lead to the decision.

The final structure of the subsystems (the diagram) will be part of the system documentation.

Modification History		
Stefan Versluys	17-02-2015	Initial discussion formulation
Guus Bonnema	20-02-2015	Transformation to decision document

Contents

1 Problem background and description

Background During the planning phase of the project, the customer requested that the designer be integrated with the verification tools in a way that should be transparant to the user. Seamless integration should be the appearance, whether or not the technical integration would be seamless.

For that reason the team experimented with different forms of integration and finally reached the stage where we could formulate the alternatives that we deem viable and decide.

This document describes in summary which alternatives we compared, what we decided and what our motivation was.

Description and alternatives Once we decided to switch to QT as the user interface toolkit, we were confronted with 3 different technologies that QT uses to implement a user interface: GraphicsView, QTQUICK 1 and QTQUICK 2. The decision at the time was to use QTQUICK 2 and the related replacement for javascript QT (referred to as QT 2).

Immediately after this decision we started studying the interface between QT and C++ in order to interface the XMD subsystem (the designer) with the VT subsystem (verification tools). The experimentation that followed lead us to three alternative implementations that we partially implemented before reaching the conclusion formulated below.

We considered the following alternatives:

C++ parser In this alternative we communicate from the designer to a mapper class that catches each signal for creation of a component on the designer and translates it into the currently existing structure (`xmas.h`). This implies complete and integral coupling of the XMD subsystem and the VT subsystem through this mapper class. Also, the flat json designed in previous activities were to be amended for the new designer capabilities including graphical data requirements.

Partial C++ parser and javascript parser This alternative implies two structures that contain the network design: one for the designer and one for the VT subsystem which is the existing `fjson` structure. This implies translation of one datastructure to another from the mapper class but limits the exposure of graphical needs of the designer to the verification structure.

Full javascript parser This alternatives implies two structures as well, but uses the `fjson` text stream as interface between the two subsystems. This implies that there is no other coupling between the XMD subsystem and the VT subsystem and no modifications to the VT system are necessary other than those needed to run and compile on all specified platforms.

2 Decision and motivation

Due to the massive advantages that the loose coupling of the alternative *Full javascript parser* offers, we decided in favour of this alternative.

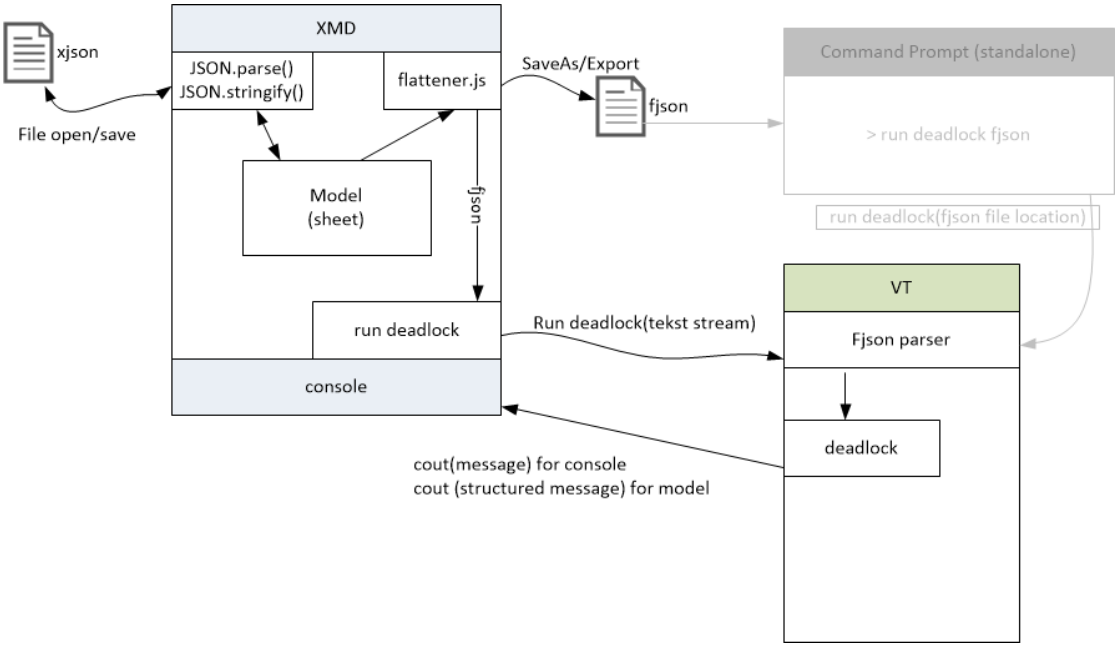

Figure 1: Diagram for *Full javascript parser*

3 Implications for future changes

The decision implies that VT development is relatively independant of designer development. Any changes to the interface of the `fjson` structure impacts both subsystems, but all other changes are independent.

Additionally we still need a mechanism to start and stop verification tools, track the error responses and we need to couple the verification tools executables into the designer without creating a development dependency.

4 Details of the original discussion for VT - XMD integration

Subject for discussion This discussion concerns the way verification tools (VT) can be integrated into or interact with the Xmas Model Designer (XMD). There are several options but not clear which is the best, because of conflicts in their requirements & specifications, additional risks, increase of complexity and fade of basic rules concerning separation.

vt requirements & specification which affect integration decision:

- The vt's are written in pure c++ because they can run on platforms without any specific software tools.

- vt's their mayor key is performance.
- The vt parser can handle a flat json structured file.
- The vt parser cannot handle composites.
- The vt parser feeds the vt model and has no GUI properties.
- The vt's can be started and get their input via the command prompt. (standalone)
- The vt's can be started and get their input via the designer tool.
- The vt's structure is complex because it has to execute an algorithm and has to know about the message details plus its focus is performance and not maintainability.
- The vt's give feedback via the standard output

xmd requirements & specification which affect integration decision:

- xmd is written in qml/c++ with Qt classes which gains development speed, fancy GUI, easy scripting.
- xmd must handle composites.
- xmd its major key is maintainability
- Composites can be inside other composites and implies a hierarchical structure
- xmd needs a hierarchical json parser
- xmd needs GUI specific component properties like x,y,orientation
- xmd doesn't need vt specific component model data structures.
- xmd must be pure GUI and doesn't need to know about complex vt structures
- Ideally vt integration in xmd must be purely interaction, as if they are one thing for the xmd user but technically totally separated.
- xmd must show the vt's feedback in the console (normal text message)
- xmd can show the vt's result in the model (structured text message)

Team:

- vt's are out of scope.
- Modifying vt's parser and the structure to pass additional data is a risk:
  - planning : complexity gains time?
  - focus is xmd : loss of vt requirements/correctness?
  - Because of OU changes, Bernard cannot be part of the team as developper.
  - xmd interaction is done through the whole structure and makes vt and xmd very dependent from each other.
  - Because of this dependence a change at the vt side can harm xmd and vice versa and breaks the maintainability rule. The same holds if changing xmd which can decrease vt its performance.
- Using the vt parser and underlaying data structure extends its complexity into xmd which conflicts with xmd's maintainability requirement.
- Bernard: use only one data model, easier to maintain.
- Bernard: vt has a parser you can use it.
- Freeks concern of diving too deep into the vt's because of too many risks and getting out of scope. (see email)

5 Options

vt pure c++ parser only:

- +/+ only one file type for XMD and VT
- +/- one parser and model : not an advantage see remarks (*)
- -/- flattening must be done at VT side in pure c++
- -/- XMD depends on the VT parser and complex VT structure for mapping.
- -/- increased exposure to change issues between VT and XMD.
- -/- VT structure mixed with GUI parameters (x,y,orientation,scale,..)
- -/- VT parser and structure modification = risk + out of scope
- -/- current fjson files not compatible anymore = loss of test references
- -/- composites need to be handled/flattened in VT parser

- -/- handling composite file locations, XMD vs. VT standalone on a pure platform
- -/- id references of components or ports can become inconsistent.

(*)The advantage of having a parser already is not an issue because in javascript a parser is one line of code. The disadvantage of having to modify two parsers /models isn't an issue because in javascript, modifying is just adding a JSON field. If vt and xmd are independent, a modification of the xmd parser doesn't lead to a vt parser modification if it's just about a GUI parameter. A flattener only pick items which are relevant for the vt's.

xmd Javascript parser + vt dependent:

- +/+ XMD can have its own json structure with GUI specific parameters
- +/+ flattening can be done outside VT, no need to use pure c++
- +/+ fjson stays compatible
- +/+ no need to modify VT to handle composites (parsing+flattening)
- +/+ composite handling not necessary in VT for this project.
- -/- two file types , one hierarchical json and one fjson. Same as the current WickedXmas.
- -/- mapping for flattening is still done via VT parser/data, meaning complex VT data structures into flattener.
- -/- a designer must hand over flat files to a verifier using VT as stand alone.

xmd Javascript parser + vt independent:

- same advantages as previous
- +/+ flattening done totally independant in e.g. Javascript , no need to know VT data structures. No mapping.
- +/+ XMD and VT are completely independent, only the fjson structure is common.
- +/- flat structure in XMD must match VT flat structure , but in javascript very small modification. Much easier than adopt mapping to complex datastructures.
- +/+ flattening in the design tool can be used to see if there are no cycles before these are send to de VTs
- +/+ very low risk, only the fjson structure must match
- +/+ no need of complex mapping code in designer, pure compact, robust and easy javascript

- -/- two file types xmd + fjson
- -/- XMD needs and SaveAs or export function to fjson , so VT s can run models as standalone

Figure 1 is a setup where VT and XMD are totally independent but can interact with each other as if they are one thing. By this XMD can keep its maintainability and apart from minor changes VT stays untouched. The fjson structure is the only key they have in common.

Example of JSON in javascript:

```
function jsonTest(){

    var object = {
        "VARS":[],
        "PACKET_TYPE":{},
        "COMPOSITE_OBJECTS":[],
        "NETWORK":[],
    }

    object.NETWORK.push(
                {
                    "id":"q1",
                    "type":"queue",
                    "x":100,
                    "y":50,
                    "orientation":90,
                    "scale":0.5,
                    "outs":[],
                    "fields":[{"size":2}]
                }
                )

    var jsonOrder = ["VARS","PACKET_TYPE","COMPOSITE_OBJECTS","NETWORK","id","type","x","y","orientation","scale","outs","fields","size"]

    var jsonFileText = JSON.stringify(object,jsonOrder,1)

    //throws exception if incorrect json
    var objFromJsonFile = JSON.parse(jsonFileText);


    console.log("=============================================================================================")
    console.log("= Json text created from object")
    console.log("=============================================================================================")
    console.log(jsonFileText)
    console.log("=============================================================================================")
    console.log("= Object created from JSON text")
    console.log("=============================================================================================")
    console.log("id=" + objFromJsonFile.NETWORK[0].id
              + ", type=" + objFromJsonFile.NETWORK[0].type
              + ",(x,y)=(" + objFromJsonFile.NETWORK[0].x + "," + objFromJsonFile.NETWORK[0].y + ")"
              + ",fields.size=" + objFromJsonFile.NETWORK[0].fields[0].size
              + ",...")
}
```

Figure 2: Javascript json example

Figure 2 is a javascript function creating a json structure which is compatible with the current fjson.

Figure 3 result of executing previous javascript function.

6 Conclusion

The third option, meaning xmd completely independent, is the most appropriate choice. It has a low risk and maximum independancy, the latter is important to prevent fading their major keys into each other, certainly because these keys are conflicting. During the last meeting with our customer, about reusing the vt's parser and use one data

```
QML debugging is enabled. Only use this in a safe environment.
qml: ================================================================================
qml: = Json text created from object
qml: ================================================================================
qml: {
 "VARS": [],
 "PACKET_TYPE": {},
 "COMPOSITE_OBJECTS": [],
 "NETWORK": [
  {
   "id": "q1",
   "type": "queue",
   "x": 100,
   "y": 50,
   "orientation": 90,
   "scale": 0.5,
   "outs": [],
   "fields": [
    {
     "size": 2
    }
   ]
  }
 ]
}
qml: ================================================================================
qml: = Object created from JSON text
qml: ================================================================================
qml: id=q1, type=queue,(x,y)=(100,50),fields.size=2,...
```

Figure 3: Javascript json console output

model for the whole application, was a good advice at first sight. Following that way we're now at a point where we bump into several problems. Vt and xmd are based on totally different requirements and by mixing those two applications together we create a big risk. The risk that we have to adopt vt too much that we will run out of time, we have two iterations left. Vt its source is complex, we noticed that even adding a simple parameter is not that easy and we still need to do a lot. Work like saving a file via the vt structures and parser, composites and the flattener in pure c++.

In option three, vt only need some small adjustments and xmd will become completely independent. The risk that during maintenance the customer has to modify two parsers doesn't exist. Parsing Json in Javascript is very easy compared to modification in complex mappings between xmd and vt. The same holds for the data model , there will be only one and that's the original vt data model because xmd doesn't need to know about it. Xmd only knows its graphical model as now in qml or Javascript. Having two application doing an other thing with the same kind of data without using the same libraries or whatever is a common thing. E.g. zip is such a common structure or something like Qt Creator which knows about the c++ syntax but only triggers an external compiler is very similar to this application.

What has to be done at vt if option three is chosen?

- A vt can be started with a fjson file argument. E.g. deadlock -f:2queues.fjson

- A vt can be started with a fjson text argument E.g. deadlock -t:fjson text stream out of xmd flattener
- A vt can be stopped
- A vt can send messages to the standard output redirected to xmd its log.
- A vt can send results to the xmd model as structured text E.g. cout([result] q1:5,q3:6.....)
- A vt can send its status on request E.g. deadlock -status

What we don't have to do in vt if option three is chosen?

- Writing a flattener in pure c++ which has to be integrated in the vt parser because of the composites.
- Handling composites in vt. E.g. what about the composite file locations? For a xmd user it can be different than for a vt user who's working standalone on a naked platform.
- facing the risk of getting out of scope.
- facing the risk of creating bugs in the vt software.
- facing the risk of degrading vt's performance.
- having the current fjson structure become incompatible, meaning loss of test reference.

What have to be changed in xmd?

- Writing a flattener in Javascript.
- Remove complex mapping with vt structures.
- Create a JSON object of the canvas (sheet).
- Write a process starter with text stream. (QProcess,stdin/out ? must be pure c++ compatible at the vt side)
- redirect vt its output to the log

By this we can focus on finalise and fine tune the project during our last two iterations and write documentation which e.g. describes how a customer can add a new kind of primitive in xmd javascript or how a user can change fjson structure with the JSON parser from javascript.

Of course the choice of our customer has the highest priority but at least the customer must be aware of these consequences.