Visual Architecture

Program Documentation

Introduction

In the current digital age, computers are very complex; possessing numerous intricate components. In order to keep abreast with emerging software trends, there is a growing need for developers to understand what goes on 'under the hood' of a digital system.

With the Visual Architecture plug-in a new outlook into the representation of low-level programming can now be explored; as it enables the virtual creation of simple digital computer components. Due to the extensive nature of Eclipse plug-ins, it is hoped that future implementations of this tool will be used to demonstrate more complex systems.

The Visual Architecture plug-in solution is a framework in Eclipse that allows a teacher to create a diagrammatic representation of some component architecture using ‘drag-and-drop’ graphical features. Once the architecture has been defined the students can then use this design to develop their own assembly code implementations. The student implementations are developed using the normal Eclipse IDE, powered by Xtext.

This solution also provides a Debugging feature that enables the visualization of developed programs. It uses a ‘step-into’ and ‘step-over’ protocol to animate execution sequences in the architecture, giving the student visual feedback for each step executed.

Technical Documentation

System Overview

The system consists of several modules operating as can be seen in the diagram below:

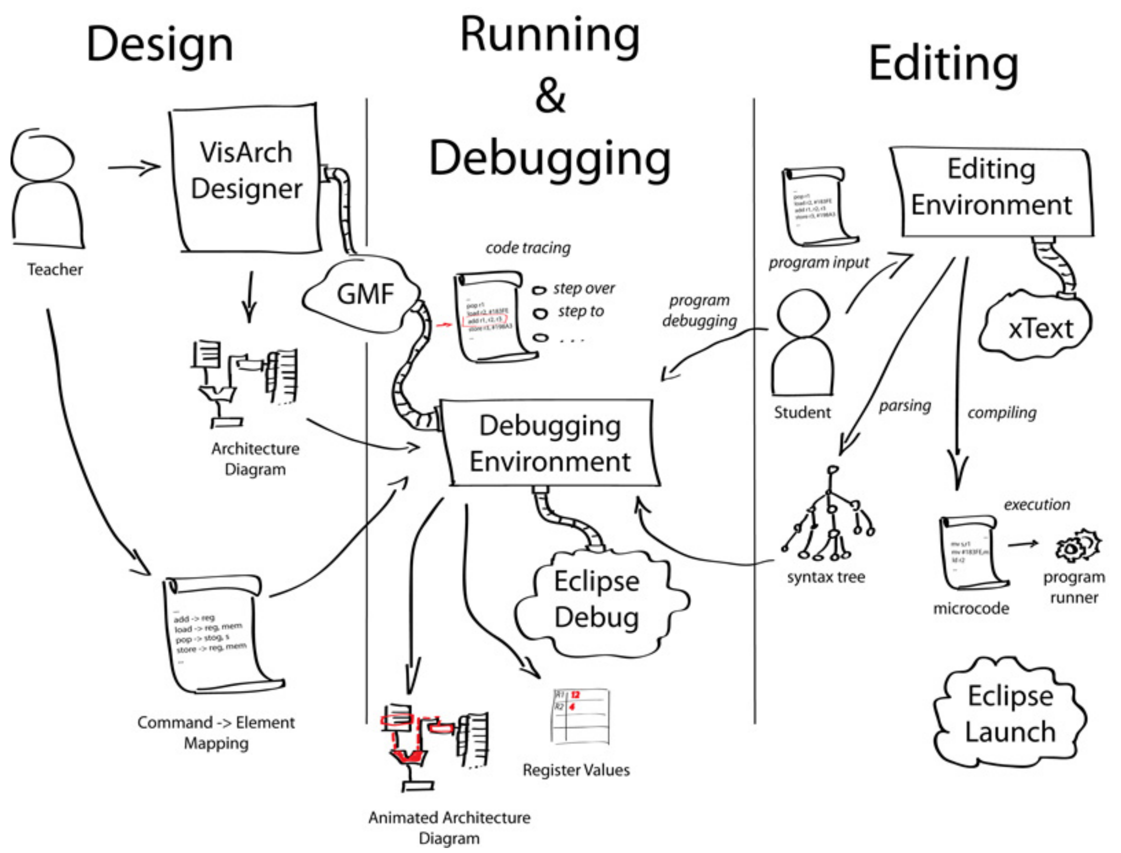


Figure 1: Demonstrates the design for module operations in the system

All of these modules are separate Eclipse plug-ins (some of them even consist of more than one plug-in) that communicate between themselves using object references, extension, points, events etc.

A Visual Architecture system is considered to be an Eclipse IDE containing all of our plug-ins. In the next few sections, these modules will be discussed in more details...

Diagram Editor

This is a GMF project. This has been implemented on top of EMF and GMF. The diagram editor allows one to create a visual diagram that represents an computer architecture. A tool palette is located on the right hand side of the editor window, in which a list of diagram design elements (like Components, ALU, Register, Memory, Bus etc.) are held. These elements are represented on *Figure 2* below. The architecture component design diagram supports drag-and-drop functionality. The compartmental functionality provided by the *Component* tool supports nested implementations where Components my be grouped into categories. GMF not only reflects the compartments in the diagram but also displays the structure trees on nested items in a separate window (*Figure 3:Node nesting in the tree-viewer)*.

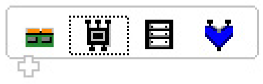


Figure 2 :Palette tools available in the design editor.

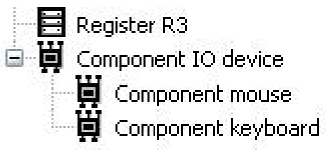


Figure 3: Shows two nested components in the IO device, as displayed in structured tree supporting editor windows of GMF

As a compartment is in itself an element, it contains all the functionality of regular components as well as its encapsulating abilities. It is important to note however that connections made to compartment figures must be targeted at the top and bottom sections of the component (and not the center).

The GMF tool utilizes a feature referred to as the GMF runtime environment that supports diagram saving and retrieval, node modification and style change; extra drawing objects; and reordering and arranging nodes and connection.

The diagram below illustrates an example of a sample diagram that could be created using the Diagram Editor.

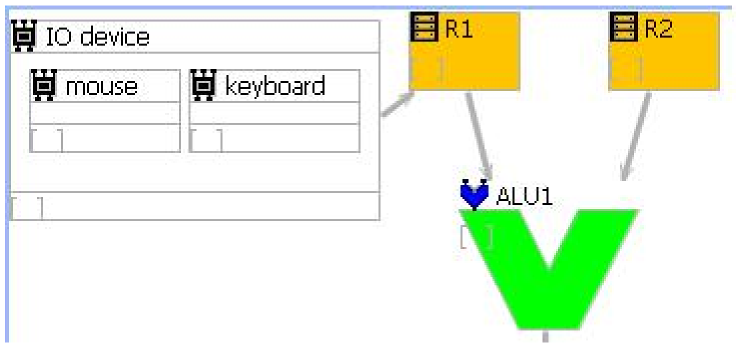


Figure 4: Sample GMF architecture

Architecture Mapping Editor

This is an X-text project that generates a grammar for '*.mapping' files. In this file you make a list of all the commands and with each command you make a list of elements in the model that are working with it. So for example: LOAD - Memory, Mem_Addres, busMem, ALU;
Every line starts with an assembly command, followed by ' - ' (space, bar, space) followed by one element. Possibly there could be more elements involved, these are added behind the first one as ', *element*' (comma, space, element). The line is always ended with a ';'.

For future development could be looked into adding clock-cycles. So what should be lit up in the first clock-step and what in the next. Also there is no problem with using one assembly command more then once and it is not clear what this would mean. Also it would be more elegant to link ‘DiagramElement’ to the real model so that you can only use elements that are in the created model. Another feature would be to use the parameters of the code as well. For example if you have ‘DEFINE Number, Register’, it would be nice that ‘Register’ gets highlited, but not the rest of the registers. Now you can only hard code which register will, so probably all.

The code is included in Appendix B.1

Assembly Editor

This is an X-text project that generates a grammar for ‘*.assembly’ files. In this file you write regular assembly code. So every line consists of a command followed by their parameters. Parameters are either registers, memory or numbers. Memory starts with M followed by a positive number and registers star with R followed by a positive number. Numbers are only allowed to be positive.
There are four assignments implemented. Those are Load, Store, Define and Add. Load is a command that is written as a line that starts with ‘LOAD’ followed by a memory, a comma (‘,’) followed by a register. The other commands look very similar. First the command in capital letters followed by the parameters divided with a comma.

For future development could be looked into adding more assembler assignments, allowing to use negative numbers with defining and make those numbers only as long as an integer.

The code is included in Appendix B.2

The Debugger

The debugger is the module of the program connecting the assembly programing environment with the diagram of the computer architecture.

The debugger consists of three main parts:
1. Assembly launcher
2. Assembly debugger
3. Assembly interpreter

The mission of the launcher is to pick up the configuration from the user (either through the configuration tab view where it’s manually entered or the default values when using a launch shortcut). After that the launcher creates an interpreter and a debug target and launches them.

The debugger works by implementing the many interfaces of the debug model:

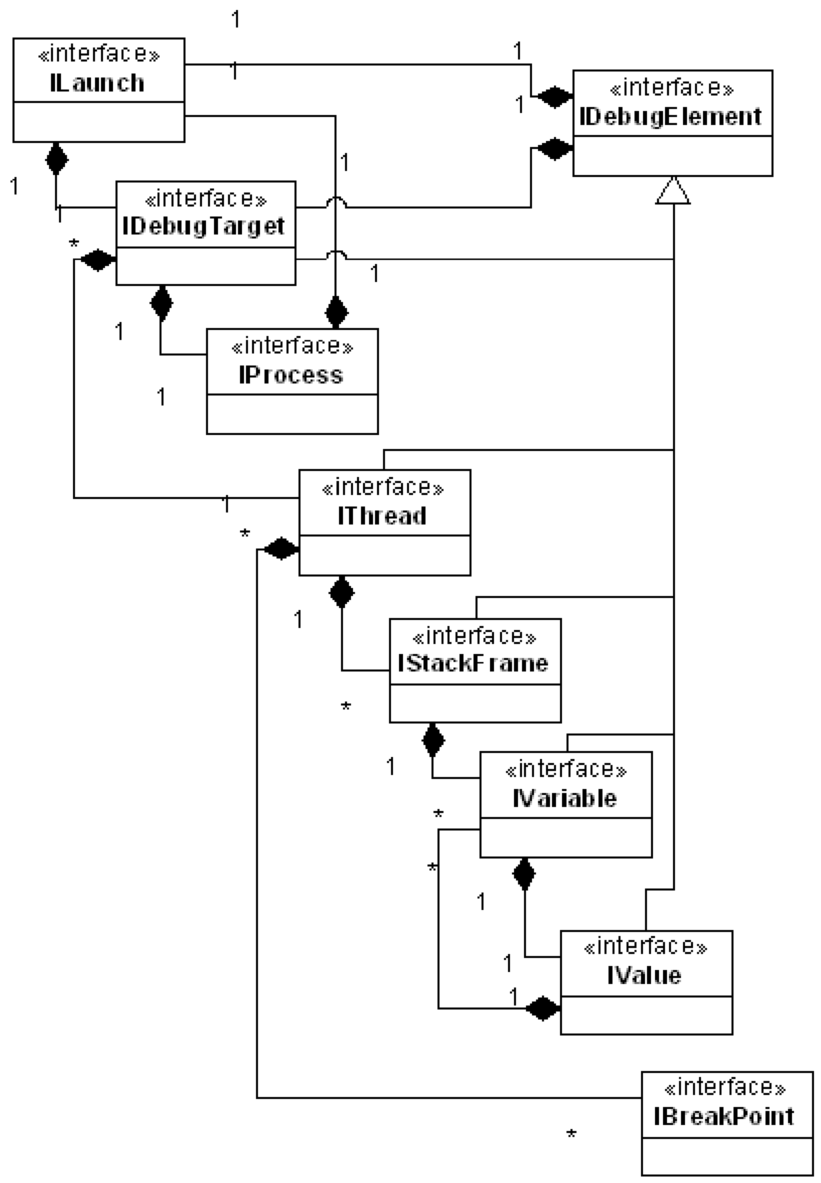


Figure 5: How the Debugger Works

It communicates with the Eclipse view using different debug events. All the main functionality ends up at the AssebmlyDebugTarget which delegates the real work of interpreting command by command to our Interpreter, a separate plug-in.

The interpreter has information from the launcher about the file that is being debugged and is notified by the debug target when the user wants to step to the next command. It can then make the real operations using it’s virtual registers, memory etc. - the whole virtual machine.

Also, at this point the interpreter could notify the diagram (it knows the resource URI, from the launch configuration) to highlight specific parts of the diagram - helping the user see what hardware is concerned with this concrete assembly command. To achieve this it would need a mapping file (for which it also knows the URI) which it can easily parse using xText.

Microcode Generator

The Microcode Generator is build in Acceleo. It uses the VAmodel and the file ‘program.assembly’. Then it generates a new file ‘Microsteps.micro’ which is a transformation of ‘pragram.assembly’.
For all the assembly commands it writes some new lines like in the table below.

Assembly code	Micro code
LOAD M1, R1 into	Move M1, Memory_address Set Memory.Read Move Memory_out, R1
ADD R1, R2, R3 into	MOVE R1, ALU_argument1 MOVE R2, ALU_argument2 ADD MOVE ALU_result, R3
STORE R3, M3 into	Move R3, Memory_in Move M3, Memory_address Set Memory.Write

So one single LOAD command will be replaced by 3 lines of microcode and one empty line. Also the arguments are copied. So if you have ‘LOAD Mem, Reg’ it will be changed into ‘Move Mem, Memory_address’.

Further development could look into using the mapping file to make a better microcode. Also could be thought about clock-cycles and how to make a clear document that states when a next clock-step is needed.

The code is included in Appendix B.3

Usage Instructions

Installation

Download the Visual Architecture deployable plug-ins from the project webpage - http://

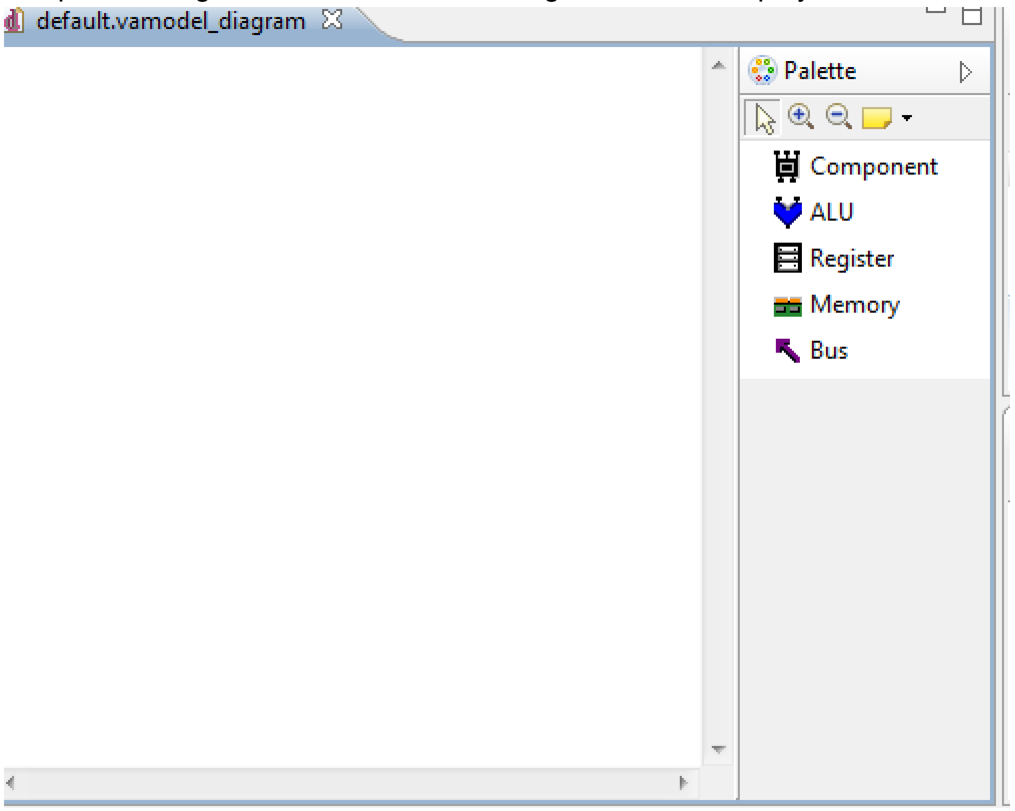
www.fer.hr/rasip/dsd/projects/visual_architecture/documents and either extract the .jar files from the zip archive to your Eclipse plugins folder (/path/to/Eclipse/plugins) or import them to your workspace and run another Eclipse development instance from Eclipse (Alt+Shift+X, E).

Anyway you chose, you should now have a running Eclipse platform with our plug-ins installed inside and can use all the Visual Architecture features.

Teacher’s Tool

The teacher’s tool is created using a project containing the diagram editor and the architecture mapping file. Once one loads the eclipse environment intended for use as a teacher’s tool, a VA model diagram and assembly file (that supports Xtext can be included).

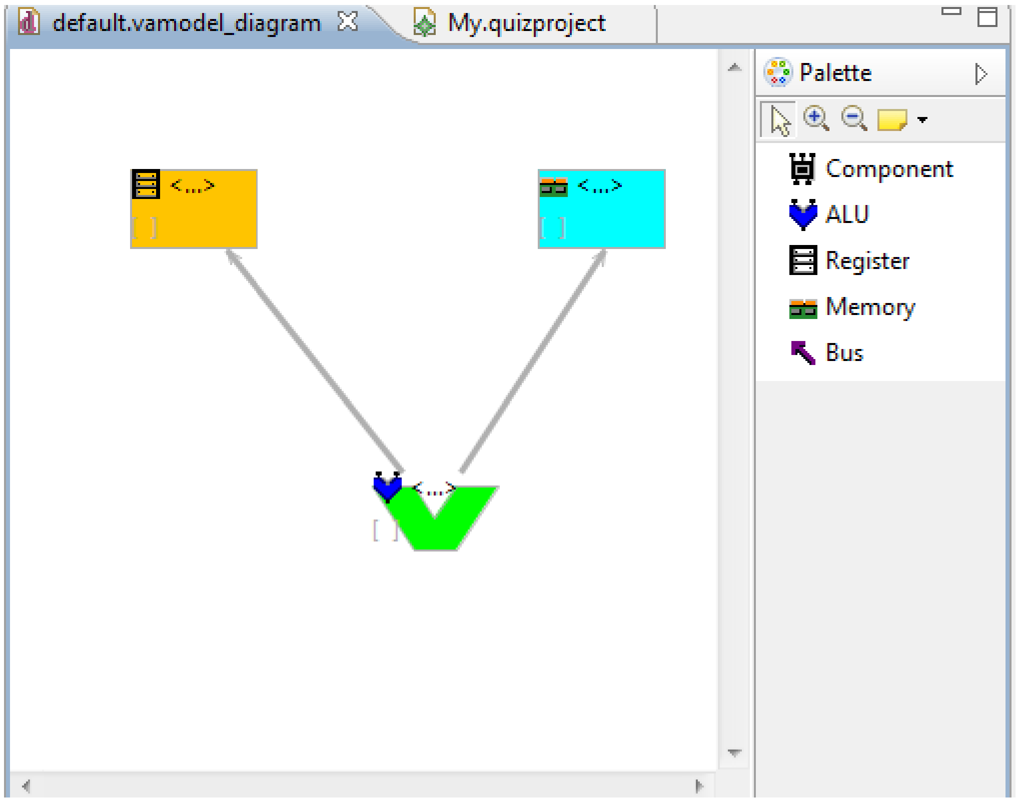
Steps to obtain the window shown below include:
1. Create a new project, or folder
2. Add a VAmodel from File-> New -> Example
3.In the wizard that appears select *VAmodel Diagram* and the file that will contain the model.
3. Upon clicking Finish, the VAmodel design window is displayed.



Creating a computer architecture diagram

Elements can be *dragged and dropped* from the tool palette (using their default dimensions), or *drawn* onto the canvas (to control their resultant sizes). Additionally, the GMF environment provides an extra means of adding nodes by clicking on the canvas area to display a mini toolbar of node elements. The teacher's tool does not dictate the architecture model design however it does restrict the connections between two objects in the same direction.

For instance, a sample architecture could resemble the figure below and multiple links between the ALU and Register or Between the ALU and memory are prohibited.



Other than the context menus supported by the editor, the diagram file also includes a menu called *Diagram* used to edit the figure further as per required.

Writing an architecture mapping file

Once you open the part where you can make a mapping file, you have to create a new file. The extension of this file should be '.mapping'. If you do so, a question will be prompted that asks if you want the x-text-grammar be applied to this file. Say 'Yes' and then you can start writing your mapping.
The idea is that you write one line for every assembly command. After this command you write a dash (' - ') and then a list of hardware elements (Memory, registers, ALU, etc.) that are divided by a comma. You end the whole line with a semicolon (';').

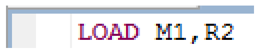
E.g.:
DEFINE - Registers;
LOAD - Memory, Registers;
ADD - Registers, ALU;
STORE - Registers, Memory;

Student's Tool

Writing an assembly program

Once you open the part where you can make an assembly file, you have to create a new file. The extension of this file should be '.assembly'. If you do so, a question will be prompted that asks if you want the x-text-grammar be applied to this file. Say 'Yes' and then you can start writing your assembly code.

Writing assembly means that you write one command on every line. You start with the name of the command, followed by the parameters (mostly two or three) which are divided by a comma (',').



```
LOAD M1,R2
```

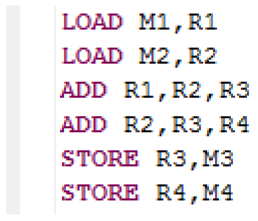
The following commands can be used:
DEFINE number, memory
LOAD memory, register
ADD register1, register2, register3
STORE register, memory

Here you should change *'number'* by a valid number, *'register'* by a valid register name and *'memory'* by a valid memory name.

Valid number is a positive integer value. Valid register name is letter 'R' fallowed by a positive integer value and valid memory name is letter M fallowed by a positive integer value.

'DEFINE' command will move a number defined by 'number' to memory location defined by 'memory'.
'LOAD' command will move a number inside memory location defined by 'memory' to register location defined by 'register'.
'ADD' command will add together numbers inside register locations defined by 'register1' and 'register2' and put the result inside register location defined by 'register3'.
'STORE' command will move the number inside register location defined by 'register' to a memory location defined by 'memory'.

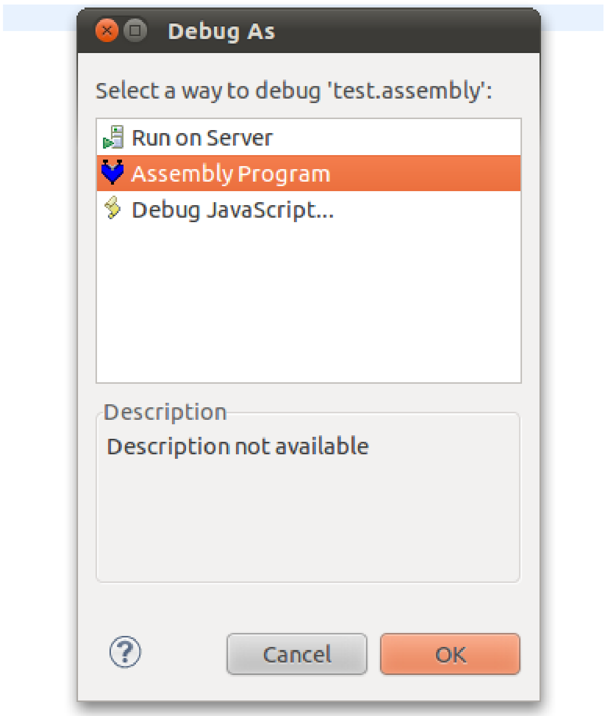
Fallowing is the example of some of the correctly written commands u can use.



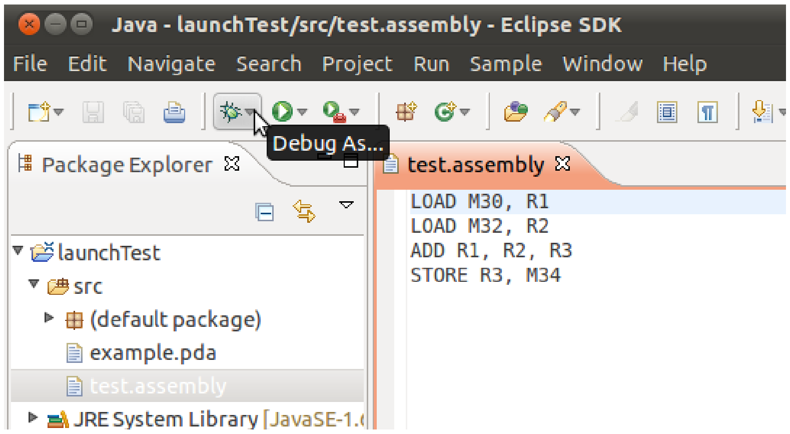
```
LOAD M1,R1
LOAD M2,R2
ADD R1,R2,R3
ADD R2,R3,R4
STORE R3,M3
STORE R4,M4
```

Debugging

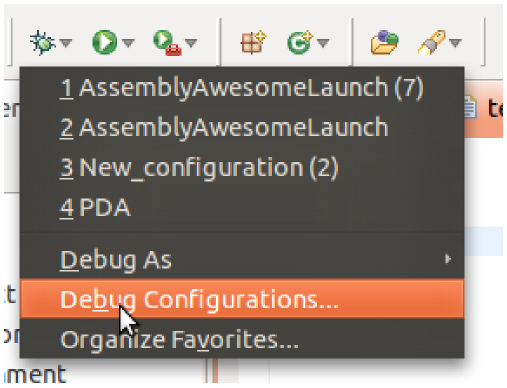
After you have written a valid .assembly file, while it's in focus clicking the green bug item to start the debugger should open a view where “Assembly awesome launch” can be selected. This will create a new configuration with the default settings and the .assembly file that was selected at the moment as the file to be launched.



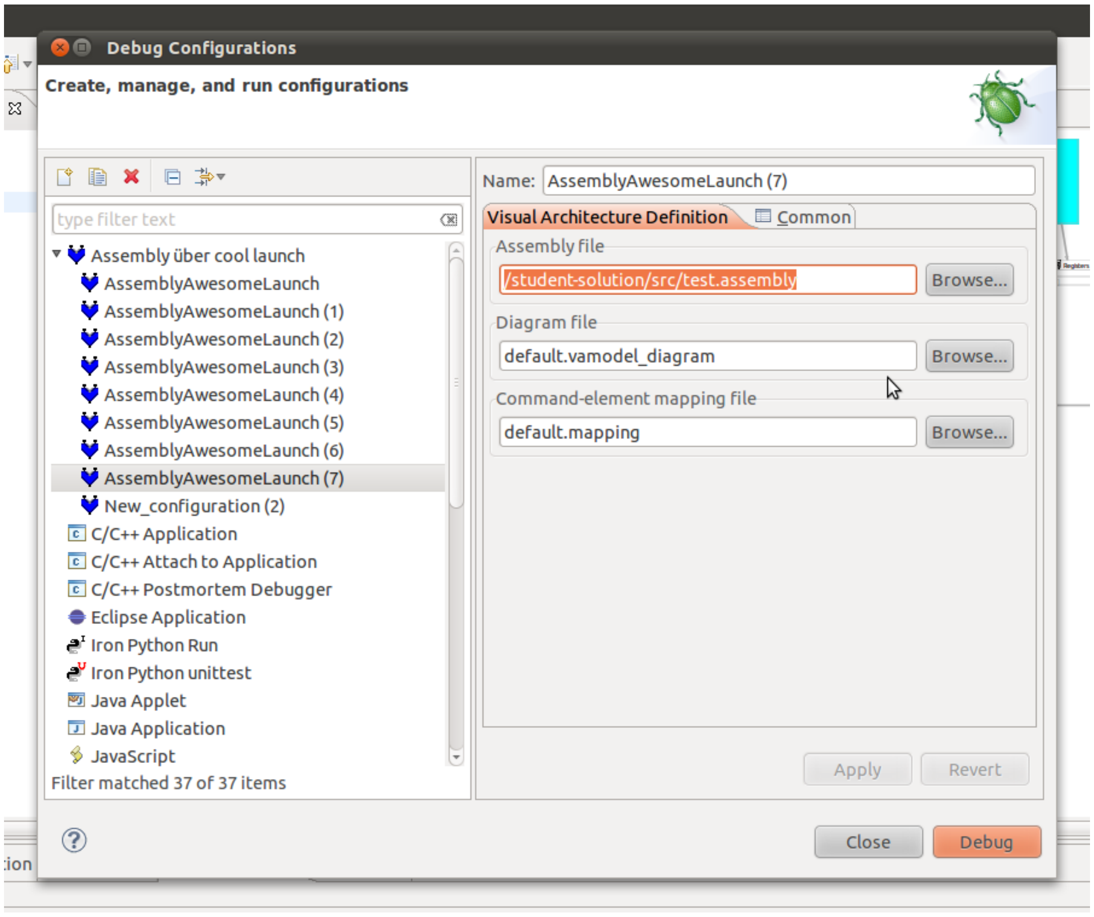
This configuration can be edited by clicking the debug dropdown menu (the small arrow next to the bug icon) and choosing debug configurations.


Java - launchTest/src/test.assembly - Eclipse SDK
File Edit Navigate Search Project Run Sample Window Help
Debug As...
Package Explorer
test.assembly
launchTest
src
(default package)
example.pda
test.assembly
JRE System Library [JavaSE-1.

```
LOAD M30, R1
LOAD M32, R2
ADD R1, R2, R3
STORE R3, M34
```



A configuration explorer will open and there should your configuration be, under the Assembly launch section. By selecting it in the right side window you should be able to edit any paths (for example selecting a different mapping or diagram file).



Debug Configurations

Create, manage, and run configurations

type filter text

- Assembly über cool launch
  - AssemblyAwesomeLaunch
  - AssemblyAwesomeLaunch (1)
  - AssemblyAwesomeLaunch (2)
  - AssemblyAwesomeLaunch (3)
  - AssemblyAwesomeLaunch (4)
  - AssemblyAwesomeLaunch (5)
  - AssemblyAwesomeLaunch (6)
  - AssemblyAwesomeLaunch (7)
  - New_configuration (2)
- C/C++ Application
- C/C++ Attach to Application
- C/C++ Postmortem Debugger
- Eclipse Application
- Iron Python Run
- Iron Python unittest
- Java Applet
- Java Application
- JavaScript

Filter matched 37 of 37 items

Name: AssemblyAwesomeLaunch (7)

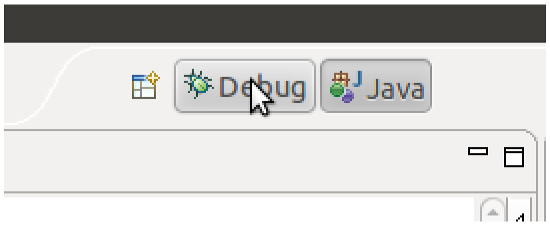
Visual Architecture Definition | Common

Assembly file: /student-solution/src/test.assembly

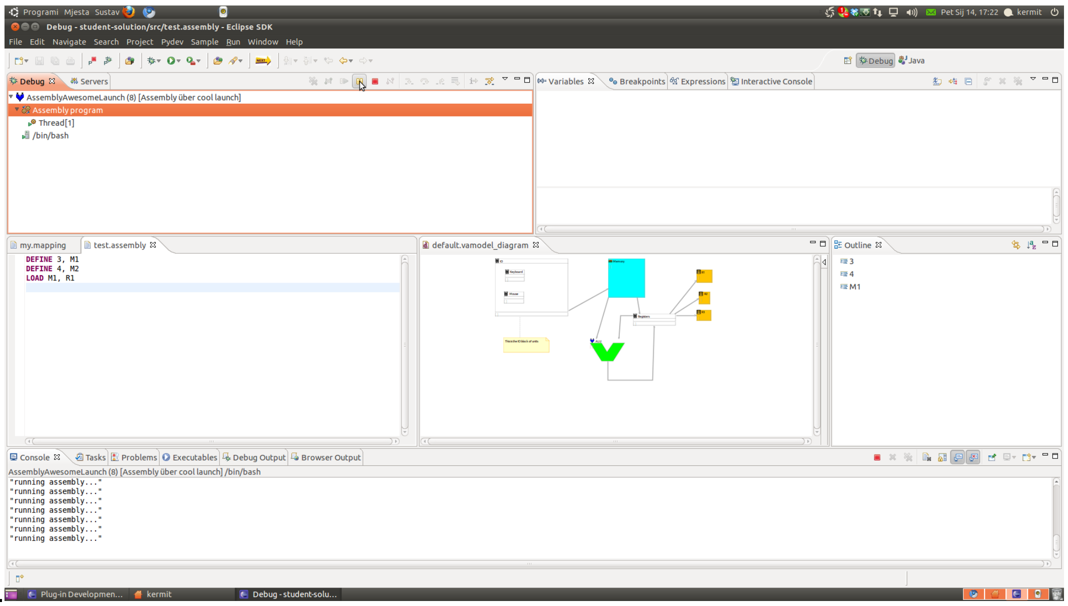
Diagram file: default.vamodel_diagram

Command-element mapping file: default.mapping

After the debugger is running you can switch to debug perspective.

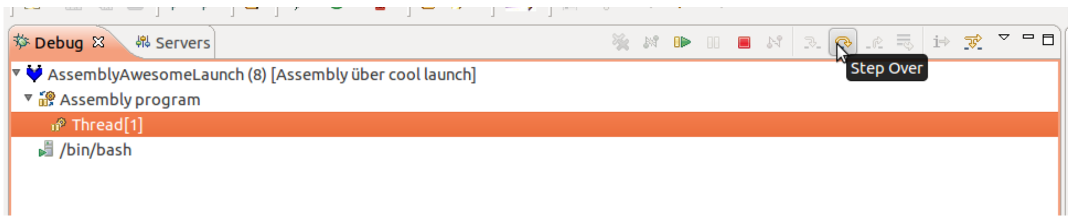


There you can see the running thread that can be suspended. Upon clicking suspend, a “step over” button will

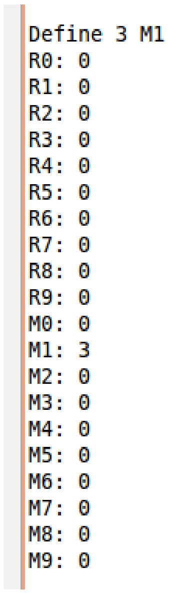
appear.


```
DEFINE 3, M1
DEFINE 4, M2
LOAD M1, R1
```

```
AssemblyAwesomeLaunch (8) [Assembly über cool launch] /bin/bash
"running assembly..."
"running assembly..."
"running assembly..."
"running assembly..."
"running assembly..."
"running assembly..."
"running assembly..."
```



Clicking on this button tells the interpreter to print out all the memory and register values (in the console of the host Eclipse for now) so that the user can see what his program is doing.



```
Define 3 M1
R0: 0
R1: 0
R2: 0
R3: 0
R4: 0
R5: 0
R6: 0
R7: 0
R8: 0
R9: 0
M0: 0
M1: 3
M2: 0
M3: 0
M4: 0
M5: 0
M6: 0
M7: 0
M8: 0
M9: 0
```

Generating Microcode

This part is not included in the software, but a beginning is made. Using this project is very simple. Just open the project and add a file to the project called ‘program.assembly’ with the assembly code. Then run the project as ‘Launch Acceleo application’ and then a file called ‘Microsteps.micro’ is generated with the microcode.

Further Development

Hard-to-implement requirements (on hold or dropped)

Visual architecture design application/process:

- Allowing teacher (architecture designer) to make new language for student and its microcode
- Providing tools for writing down the language grammar
- Allowing teacher (architecture designer) to design parts
- Providing tools for drawing parts

Code input and editing application/process

- Providing means of selecting the language used

Visual code running application/process

- Highlighting the diagram from the interpreter when debugging - everything is prepared to achieve this, but due to lack of time and no obvious GMF tutorial explaining this, hasn’t been implemented
- Providing tools for advanced(break-point) progressing trough lines of code
- Showing advanced debug information (registers, variables) in special window

Experimental future features

As with all graphical interfaces, there are many ways in which one can take the technology as an advantage to present something to the user in a more natural way – to shape the program according to the user, not vice-versa.

These are of course features very difficult to implement and would probably be projects on their own, but interesting none the less.

For Visual Architecture, examples of such features might be:

- **hierarchical architectures -** designing and examining architectures with a hierarchical graphical representation; where a student can select the level of abstraction in which he will watch the execution of his program (for example just the processor and memory or the inside of the processor – which register is being addressed, what wire is being used)
- **program execution animation** – the student executing his program on an architecture might see data animated as if flowing from one location to another through wires; this

would appear more natural to human eye and would be easier to follow changes

Appendix

Appendix A: List of Tutorials

EMF

http://gmfsamples.tuxfamily.org/wiki/doku.php?id=start

http://www.eclipse.org/modeling/emf/

GMF

http://gmfsamples.tuxfamily.org/wiki/doku.php?id=start

http://wiki.eclipse.org/Graphical_Modeling_Framework/Tutorial

http://www.eclipse.org/modeling/gmp/

http://www.devx.com/architect/Article/43366/1763/page/1

http://www.ibm.com/developerworks/opensource/library/os-ecl-gmf/

X-Text

http://www.eclipse.org/Xtext/documentation/latest/xtext.html

Acceleo

http://www.acceleo.org/pages/quickstart/en

Debugger

- List of all the documents about the debug framework (Eclipsecon 2006 is very interesting as it offers exercises to systematically solve and understand how the debugger works) - http://www.eclipse.org/eclipse/debug/documents.php
- Launcher - http://www.eclipse.org/articles/Article-Launch-Framework/launch.html
- Context Launching - http://www.eclipse.org/eclipse/debug/documents/launching/context_launching/Context-launching.pdf
- Debugger - http://www.eclipse.org/articles/Article-Debugger/how-to.html
- Memory View - http://www.eclipse.org/articles/article.php?file=Article-MemoryView/

index.html

Also it’s very useful to check out specific javadocs concerning the interfaces to the debug model, since they provide a lot of explanations such as: http://www.cise.ufl.edu/mirrors/eclipse/eclipse/downloads/documentation/2.0/html/plugins/org.eclipse.platform.doc.isv/reference/api/org/eclipse/debug/core/model/IDebugTarget.html

Some more information can be found in the Eclipse help system under Plug-in development, section debug support:
Platform Plug-in Developer Guide > Programmer's Guide > Program debug and launch support > Debugging a program

Appendix B: Program Code

1: X-Text Architecture Mapping Editor

```
Model:
        (elements+=Assignment)*;

Assignment:
        Command ' - ' DiagramElements;

DiagramElements:
        DiagramElement (', ' DiagramElement)*;

Command:
        Load | Add  | Store;

Load:
'LOAD';

Store:
'STORE';

Add:
'ADD';

terminal DiagramElement : ('a'..'z' | 'A'..'Z' | '0'..'9')+;
```

2: X-Text: Assembly Editor

```
grammar va.Assembly with org.eclipse.xtext.common.Terminals
```

```
generate assembly "http://www.Assembly.va"

Model:
        (elements+=Command)*;

terminal REG : 'R'('0'..'9')+;

terminal MEM : 'M'('0'..'9')+;

terminal NUMBER : ('0'..'9')+;

Command:
        Load | Add  | Store | Define;

Define:
'DEFINE' number=NUMBER ',' memory=MEM;

Load:
'LOAD' memory=MEM ',' register=REG;

Store:
'STORE' register=REG ',' memory=MEM;

Add:
'ADD' register1=REG ',' register2=REG ',' registerResult=REG;
```

3: Acceleo: Microcode Generator

```
        [comment @main /]
        [file ('Microsteps.micro', false, 'UTF-8')]
                [comment for (varName : VarType | whereToGetVarsFrom-List)/]
                [for (cm : Command | m.elements)]
                        [if cm.oclIsKindOf(Load)]
Move [cm.oclAsType(Load).memory /], Memory_address
Set Memory.Read
Move Memory_out, [cm.oclAsType(Load).register/]

                        [/if]
                        [if cm.oclIsKindOf(Store)]
Move [cm.oclAsType(Store).register/], Memory_in
Move [cm.oclAsType(Store).memory /], Memory_address
Set Memory.Write

                        [/if]
                        [if cm.oclIsKindOf(Add)]
```

```
MOVE [cm.oclAsType(Add).register1 /], ALU_argument1
MOVE [cm.oclAsType(Add).register2 /], ALU_argument2
ADD
MOVE ALU_result, [cm.oclAsType(Add).registerResult /]

                              [/if]
                    [/for]
          [/file]
```