Tool Evaluation – Lattix LDM

Mini Project

17-654 Analysis of Software Artifacts
April 2006

Team OverHEAD
Karim Jamal
Clinton Jenkins

Table of Contents

Table of Tables

Table of Figures

1. Background

1.1. Description of Tool

The tool that we analyzed is Lattix LDM. This tool is used to analyze the dependencies between software artifacts in a project. Unlike the more common dependency analysis tools that use box-and-line diagrams to show dependencies among components, Lattix LDM uses a structure called a Dependency Structure Matrix (DSM; also called Design Structure Matrix). The DSM approach to dependency analysis uses a matrix of rows and columns to show how each component depends on the other components within a project. It uses static analysis to compute the matrix. Unlike the box-and-line diagrams, DSMs scale relatively well when used with large, complex projects. For more information about DSMs, see [1]. The figures presented in later sections show examples of DSMs in action.

At the time of this writing, Lattix LDM supports dependency analysis for Java, C, and C++ projects. The projects that we analyzed with this tool were written in Java. We did not analyze projects written in C/C++ because of the extra setup and software required to do so, which is explained in a later section.

This tool comes as a stand-alone application and as an Eclipse plug-in. We used both of these as we evaluated the tool. (Eclipse is an open-source, integrated development environment (IDE) for Java. For more information about Eclipse, see [2].)

1.2. Version of Tool Used

There are a few different versions of this tool that are available. The version that we used in order to evaluate the tool was the Community Version. This version has the basic features of the tool. It does not, however, have design rules and the ability to enforce dependency constraints between different versions of the project. Nonetheless, due to the limited resources (i.e. time and personnel) that we have available to evaluate the tool, we feel that the Community Version will suffice for this evaluation. Additionally, since the Community Version is free and has no expiration, we will be able to put it to practical use within our practicum project; how we intend to use this in our practicum project is discussed in a later section.

2. Application to Software Projects

2.1. Projects

As mentioned above, we used the tool to analyze dependencies in Java projects. The breakdown of the projects we analyzed is as follows: one trivial project, four mid-sized projects, and two large, complex projects. Table 1 shows the categories and the total source-lines-of-code (SLOC) for each project.

Project Category	Project Name	SLOC
Trivial	Trivial	15
Mid-sized	A1	369
	A2	530
	A3	684
	lpsolve	509
Large, complex	Crystal2	4244
	ParkNPark	6466

Table 1. Project Categories and SLOC

As can be seen from the above tables, our evaluation covered a broad range of projects. Lattix LDM produced a DSM for each of these projects without any problems. Figures 1 and 2 show the extremes of the DSMs produced by our evaluation. Figure 1 shows the DSM for the Trivial project, which was the smallest project we analyzed. Figure 2 shows the DSM for the ParkNPark project, which was the largest project we analyzed. The tool constructs the DSM for a project by analyzing the dependencies between the class files in the project. In our case, the tool used the Java class files in the project.

The main reason for analyzing the Trivial project was to check the tool's behavior in a boundary case. We wanted to verify that, given a project with only one Java file, the tool produces the appropriate matrix. The main reason for analyzing the ParkNPark project was to determine how the tool responds to large, complex projects. As can be seen from the figures, the tool did scale well as the sizes of the projects increased.

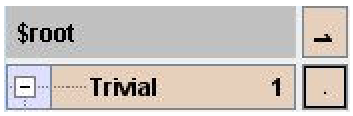


Figure 1. DSM for the Trivial Project

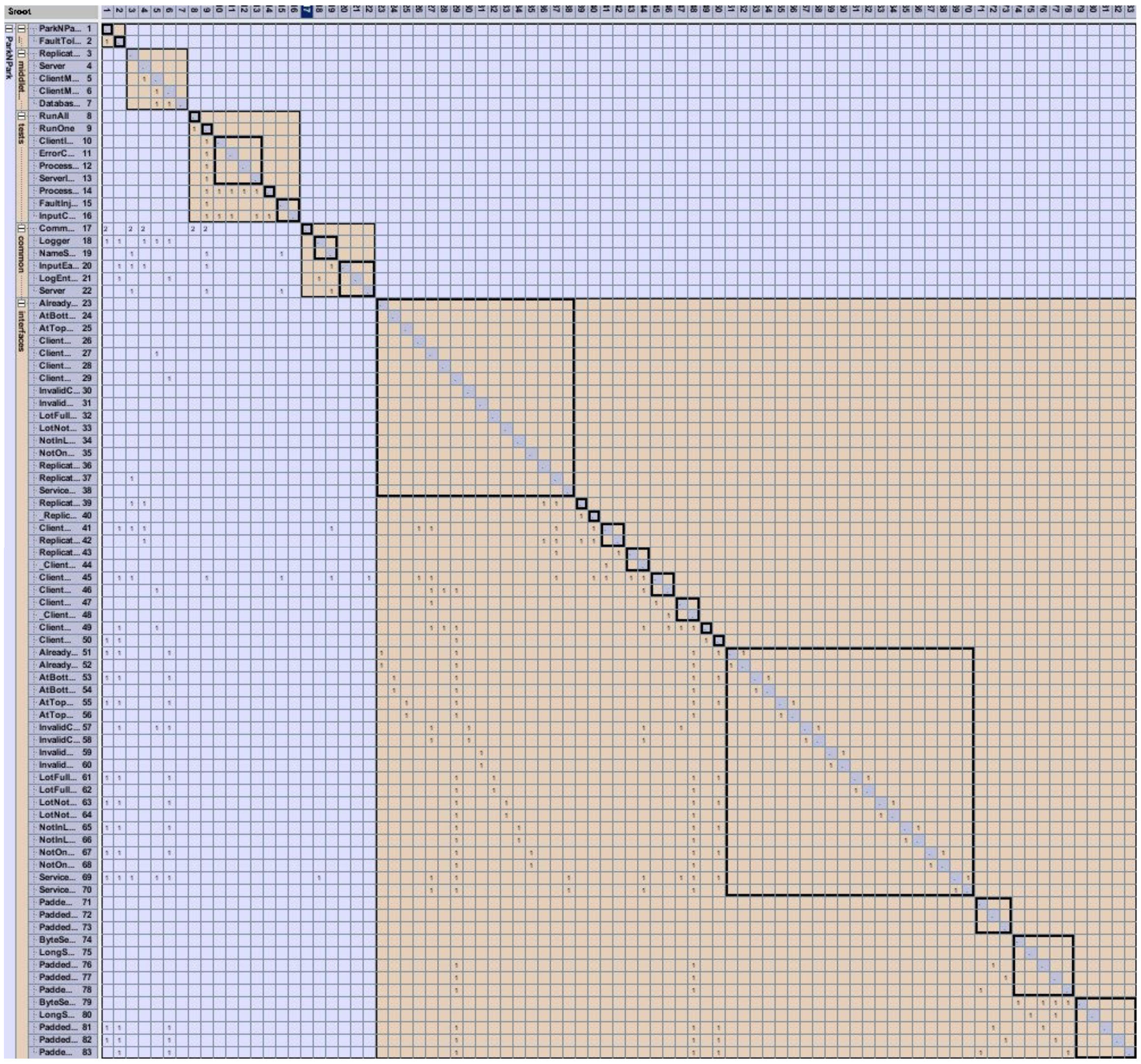


Figure 2. DSM for the ParkNPark Project

2.2. Measurements

We gathered both quantitative data and qualitative data when performing the evaluation of Lattix LDM. In terms of the quantitative data, we measured the following:

1. The number of dependencies that the tool identified correctly
2. The number of dependencies that the tool failed to identify
3. The number of extraneous dependencies that the tool identified

We followed the following process when quantifying the data.

1. We computed the DSM for the project by hand
2. We used the tool to compute the DSM for the project
3. We went through each box (i.e. each intersection of row and column) in the DSM and compared the result of the hand-computed DSM against that of the tool-generated DSM

We followed this process for the trivial and mid-sized projects. We did not perform this process for the large, complex projects due to the limited time and personnel available to perform the tool evaluation. (See Table 1 for more information about the projects in each category.)

Our results are shown in Table 2.

Project Category	Project Name	Dependency Measurements*	Count
Trivial	Trivial	Identified Correctly	0
		Failed to Identify	0
		Extraneously Identified	0
Mid-sized	A1	Identified Correctly	15
		Failed to Identify	0
		Extraneously Identified	0
	A2	Identified Correctly	16
		Failed to Identify	0
		Extraneously Identified	0
	A3	Identified Correctly	51
		Failed to Identify	0
		Extraneously Identified	0
	lpsolve	Identified Correctly	26
		Failed to Identify	0
		Extraneously Identified	0

* The ‘Identified Correctly’ measurement doesn’t include a file’s dependency on itself

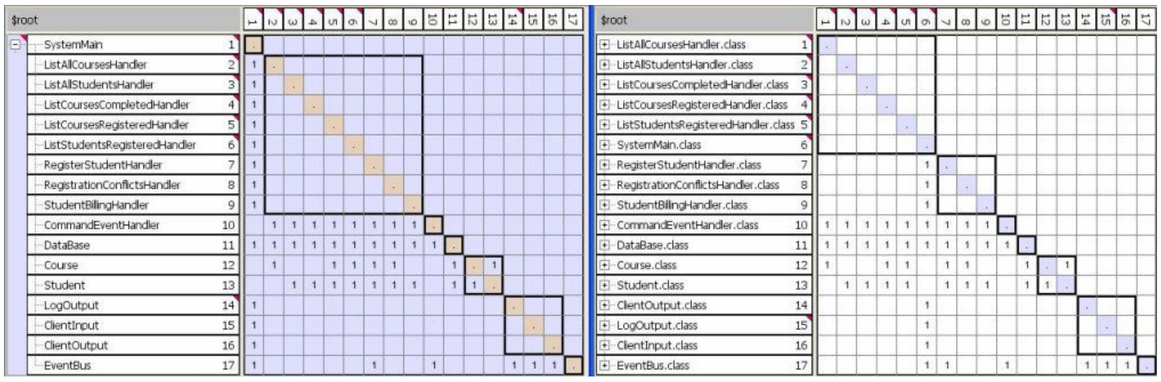
Table 2. Quantitative Measurements

As the table shows, the tool correctly computed all the syntactic dependencies within the analyzed projects. Additionally, it did not report any false positives, as can be seen in Table 1, which reports that no extraneous dependencies were found in any of the projects. Thus, since no false positives were reported, the tool seems to be precise. Additionally, since the tool found all the dependencies in the project, as compared to our hand-computed DSMs, the tool also seems to be sound. These results support the tool's claim of being able to correctly identify all of a project's syntactic dependencies. This is positive news, especially since this tool is a commercial product.

The time Lattix LDM took to compute the DSM for a project was not calculated because the tool took less than thirty seconds to perform this operation, even for the large, complex projects.

One type of qualitative data that was gathered was how well the tool arranged the Java class files in the DSM. Lattix LDM supports a hierarchical structure in the DSM. The highest level in the DSM is the project itself. The next levels are Java packages (if they are present in the project) or Java class files. The tool supports nested packages. These can be seen as the collapsible boxes in the leftmost columns in Figure 2. For each project that we analyzed (including the large, complex projects), Lattix LDM correctly computed the hierarchical structure of the project and displayed it in the DSM. The only difference between the actual layout of the hierarchy and the tool's layout of the hierarchy was that, if a Java package and a Java class file resided side-by-side in the same parent Java package or project, then the class file was put into a package named '*'. However, this is not a problem because Java packages cannot be named '*'. Additionally, this has been documented in the tool's documentation.

Another type of qualitative measurement that was gathered was that of semantic dependencies. We tried to determine how well the tool-generated DSM would capture semantic dependencies that existed within the project. We also performed experiments to check if the tool could discover semantic dependencies that we injected into the project. One such example dealt with polymorphism in the A3 project. We computed the DSM for the A3 project before and after introducing polymorphism. Figure 3 displays a side-by-side view of the resultant DSMs. For both the DSMs, the dependencies should be almost the same. However, as can be seen from the figure, this is not the case. Thus, the tool is not capable of identifying semantic dependencies very well. This is further discussed in a later section.



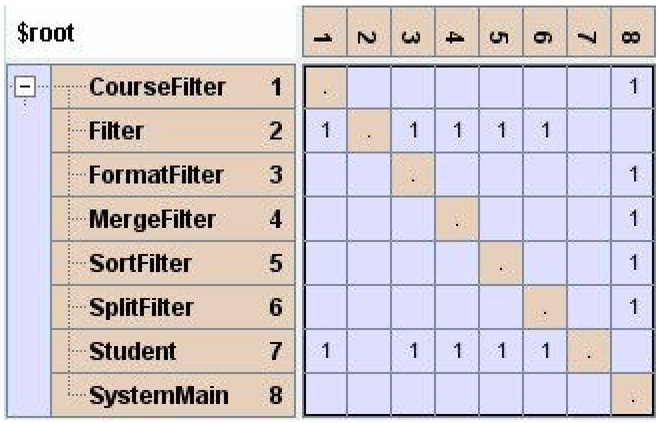
$root		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
SystemMain	1	.																
ListAllCoursesHandler	2	1	.															
ListAllStudentsHandler	3	1		.														
ListCoursesCompletedHandler	4	1			.													
ListCoursesRegisteredHandler	5	1				.												
ListStudentsRegisteredHandler	6	1					.											
RegisterStudentHandler	7	1						.										
RegistrationConflictsHandler	8	1							.									
StudentBillingHandler	9	1								.								
CommandEventHandler	10		1	1	1	1	1	1	1	1	.							
DataBase	11	1	1	1	1	1	1	1	1	1	1	.						
Course	12		1			1	1	1	1			1	.	1				
Student	13			1	1	1	1	1	1	1		1	1	.				
LogOutput	14	1													.			
ClientInput	15	1														.		
ClientOutput	16	1															.	
EventBus	17	1						1			1				1	1	1	.

$root		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
ListAllCoursesHandler.class	1	.																
ListAllStudentsHandler.class	2		.															
ListCoursesCompletedHandler.class	3			.														
ListCoursesRegisteredHandler.class	4				.													
ListStudentsRegisteredHandler.class	5					.												
SystemMain.class	6						.											
RegisterStudentHandler.class	7						1	.										
RegistrationConflictsHandler.class	8						1		.									
StudentBillingHandler.class	9						1			.								
CommandEventHandler.class	10	1	1	1	1	1	1	1	1	1	.							
DataBase.class	11	1	1	1	1	1	1	1	1	1	1	.						
Course.class	12	1			1	1		1	1			1	.	1				
Student.class	13		1	1	1	1		1	1	1		1	1	.				
ClientOutput.class	14						1								.			
LogOutput.class	15						1									.		
ClientInput.class	16						1										.	
EventBus.class	17						1	1			1				1	1	1	.

Figure 3. Semantic Dependencies for the A3 project.
(Left) DSM for the original project; (Right) DSM for the project with polymorphism

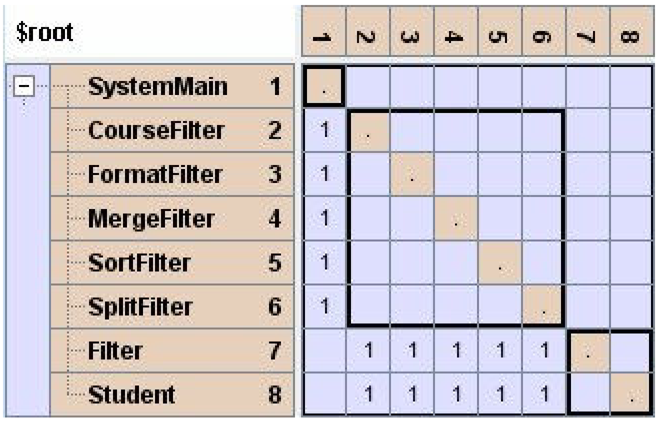
Another qualitative measurement was the quality of the tool's partitioning algorithm. The partitioning algorithm attempts to rearrange the rows in the DSM such that the DSM forms a block-triangular structure. This means that all the dependencies in the matrix either fall above or below the diagonal. If all the dependencies fall below the diagonal, then this means that a specified row only depends on rows that are above it. This can help architects determine whether or not specified architectural styles, such as layers, are implemented correctly. We used the tool's partitioning algorithm on all the DSMs that were produced by the tool. The results met our expectations. The tool was able to form a block-triangular structure in almost every case; in the cases where it could not do so, it was because a cyclic dependency existed, and thus, forming a block-triangular structure was not feasible. Figure 4 shows the original DSM for the A1 project. Figure 5 shows the DSM after the partitioning algorithm was applied. Similarly, Figures 6 and 7 show the original DSM and the DSM with the partitioning algorithm applied, respectively, on the larger Crystal2 project.

As the figures represent, the partitioning algorithm works very well, even for large, complex projects.



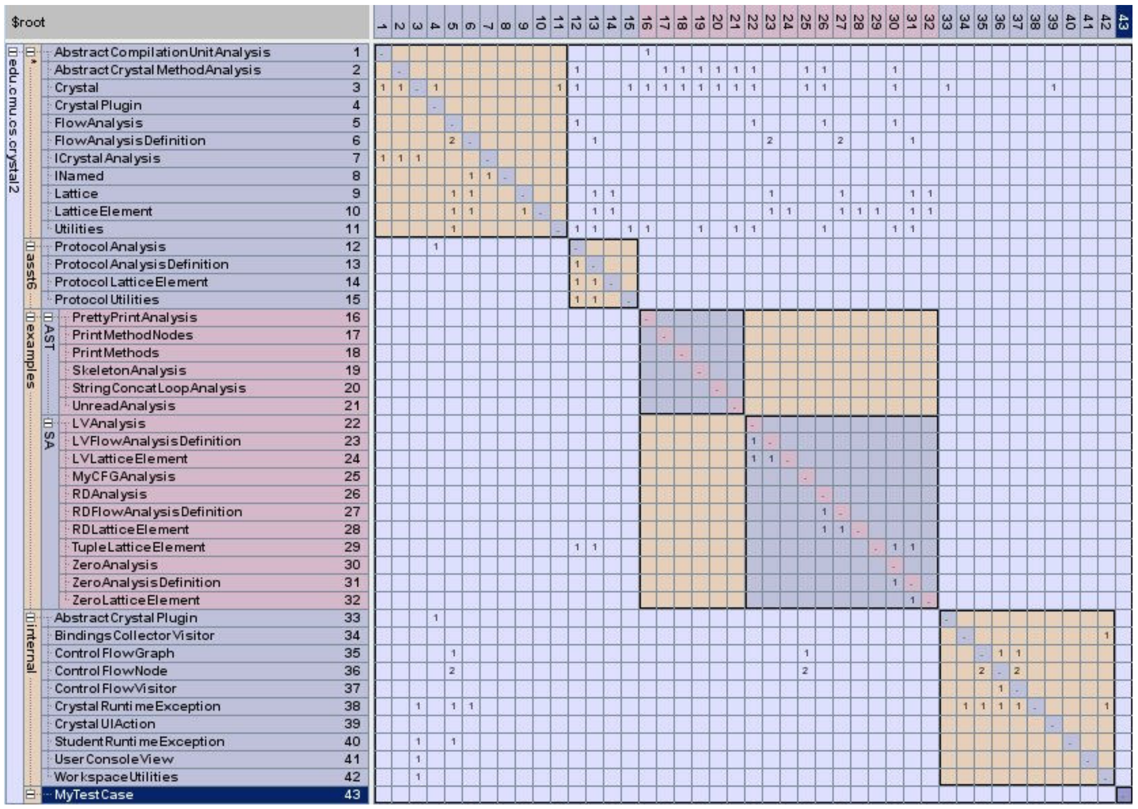
$root		1	2	3	4	5	6	7	8
CourseFilter	1	.							1
Filter	2	1	.	1	1	1	1		
FormatFilter	3			.					1
MergeFilter	4				.				1
SortFilter	5					.			1
SplitFilter	6						.		1
Student	7	1		1	1	1	1	.	
SystemMain	8								.

Figure 4. Original DSM for the A1 Project



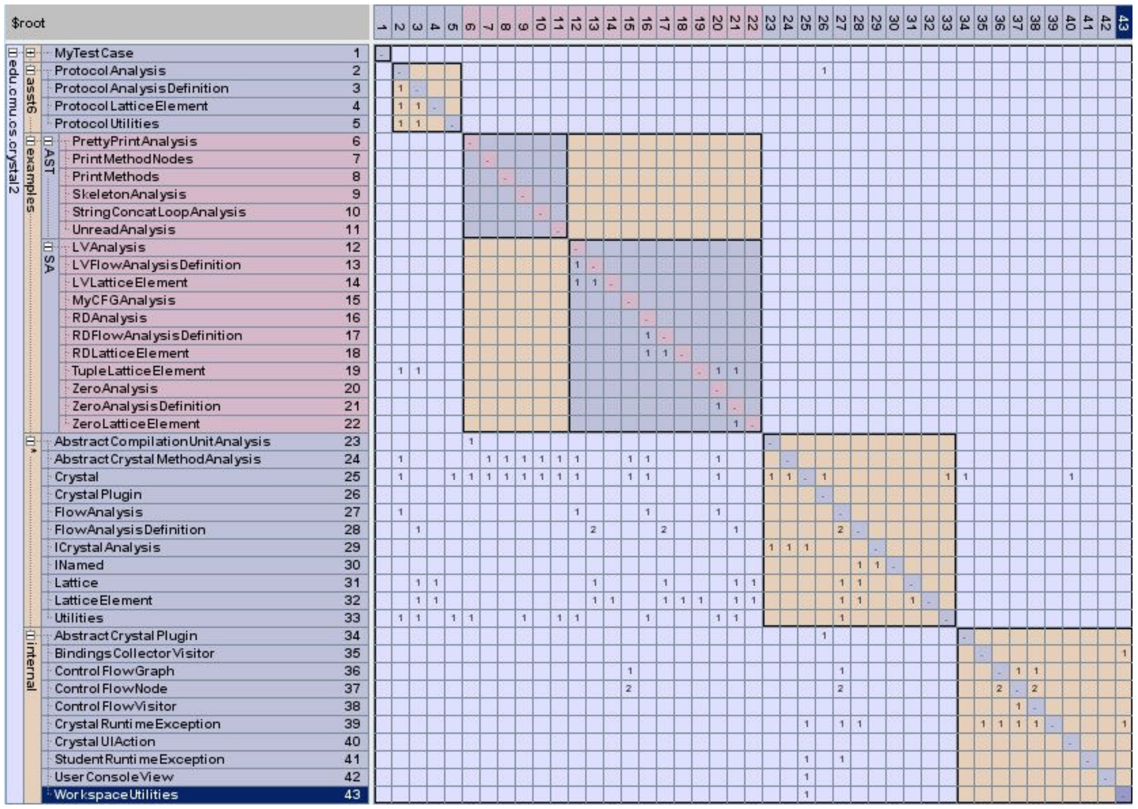
$root		1	2	3	4	5	6	7	8
SystemMain	1	.							
CourseFilter	2	1	.						
FormatFilter	3	1		.					
MergeFilter	4	1			.				
SortFilter	5	1				.			
SplitFilter	6	1					.		
Filter	7		1	1	1	1	1	.	
Student	8		1	1	1	1	1		.

Figure 5. DSM for the A1 Project after applying the partitioning algorithm



$root					1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43
edu.cmu.cs.crystal2	*		AbstractCompilationUnitAnalysis	1	-															1																											
			AbstractCrystalMethodAnalysis	2		-										1					1	1	1	1	1	1			1	1				1													
			Crystal	3	1	1	-	1							1	1			1	1	1	1	1	1	1	1			1	1				1			1						1				
			CrystalPlugin	4				-																																						
			FlowAnalysis	5					-							1										1				1				1													
			FlowAnalysisDefinition	6					2	-							1										2				2				1												
			ICrystalAnalysis	7	1	1	1				-																																			
			INamed	8						1	1	-																																	
			Lattice	9					1	1			-				1	1									1				1				1	1											
			LatticeElement	10					1	1			1	-			1	1									1	1			1	1	1		1	1											
			Utilities	11					1						-	1	1		1	1			1		1	1				1				1	1												
	asst6		ProtocolAnalysis	12				1								-																														
			ProtocolAnalysisDefinition	13												1	-																													
			ProtocolLatticeElement	14												1	1	-																												
			ProtocolUtilities	15												1	1		-																											
	examples	AST	PrettyPrintAnalysis	16																-																											
			PrintMethodNodes	17																	-																									
			PrintMethods	18																		-																								
			SkeletonAnalysis	19																			-																							
			StringConcatLoopAnalysis	20																				-																						
			UnreadAnalysis	21																					-																						
		SA	LVAnalysis	22																						-																					
			LVFlowAnalysisDefinition	23																						1	-																				
			LVLatticeElement	24																						1	1	-																			
			MyCFGAnalysis	25																									-																		
			RDAnalysis	26																										-																	
			RDFlowAnalysisDefinition	27																										1	-																
			RDLatticeElement	28																										1	1	-															
			TupleLatticeElement	29												1	1																	-	1	1												
			ZeroAnalysis	30																														-													
			ZeroAnalysisDefinition	31																														1	-												
			ZeroLatticeElement	32																															1	-											
	internal		AbstractCrystalPlugin	33				1																													-										
			BindingsCollectorVisitor	34																																		-								1	
			ControlFlowGraph	35					1																				1										-	1	1						
			ControlFlowNode	36					2																				2										2	-	2						
			ControlFlowVisitor	37																																			1	-						
			CrystalRuntimeException	38			1		1	1																												1	1	1	1	-				1	
			CrystalUIAction	39																																							-				
			StudentRuntimeException	40			1		1																																			-			
			UserConsoleView	41			1																																						-		
			WorkspaceUtilities	42			1																																							-	
			MyTestCase	43																																										-

Figure 6. Original DSM for the Crystal2 Project



$root					1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43
edu.cmu.cs.crystal2			MyTestCase	1	-																																										
	asst6		ProtocolAnalysis	2		-																								1																	
			ProtocolAnalysisDefinition	3		1	-																																								
			ProtocolLatticeElement	4		1	1	-																																							
			ProtocolUtilities	5		1	1		-																																						
	examples	AST	PrettyPrintAnalysis	6						-																																					
			PrintMethodNodes	7							-																																				
			PrintMethods	8								-																																			
			SkeletonAnalysis	9									-																																		
			StringConcatLoopAnalysis	10										-																																	
			UnreadAnalysis	11											-																																
		SA	LVAnalysis	12												-																															
			LVFlowAnalysisDefinition	13												1	-																														
			LVLatticeElement	14												1	1	-																													
			MyCFGAnalysis	15															-																												
			RDAnalysis	16																-																											
			RDFlowAnalysisDefinition	17																1	-																										
			RDLatticeElement	18																1	1	-																									
			TupleLatticeElement	19		1	1																-	1	1																							
			ZeroAnalysis	20																				-																							
			ZeroAnalysisDefinition	21																				1	-																						
			ZeroLatticeElement	22																					1	-																					
	*		AbstractCompilationUnitAnalysis	23						1																	-																				
			AbstractCrystalMethodAnalysis	24		1					1	1	1	1	1	1			1	1				1				-																			
			Crystal	25		1			1	1	1	1	1	1	1	1			1	1				1			1	1	-	1								1	1						1			
			CrystalPlugin	26																										-																	
			FlowAnalysis	27		1										1				1				1							-																
			FlowAnalysisDefinition	28			1										2				2				1						2	-															
			ICrystalAnalysis	29																							1	1	1				-														
			INamed	30																												1	1	-													
			Lattice	31			1	1									1				1				1	1					1	1			-												
			LatticeElement	32			1	1									1	1			1	1	1		1	1					1	1			1	-											
			Utilities	33		1	1		1	1			1		1	1				1				1	1						1						-										
	internal		AbstractCrystalPlugin	34																										1								-									
			BindingsCollectorVisitor	35																																			-								1
			ControlFlowGraph	36															1												1										-	1	1					
			ControlFlowNode	37															2												2										2	-	2					
			ControlFlowVisitor	38																																					1	-					
			CrystalRuntimeException	39																									1		1	1							1	1	1	1	-				1
			CrystalUIAction	40																																							-			
			StudentRuntimeException	41																									1		1														-		
			UserConsoleView	42																									1																	-	
			WorkspaceUtilities	43																									1																		-

Figure 7. DSM for the Cystal2 Project after applying the partitioning algorithm

3. Lessons Learned

3.1. Scope of Tool

The scope of the tool in the context of this report primarily describes the kinds of projects that Lattix LDM would be applicable to and useful for. The primary factors that will determine what types of projects Lattix LDM can be used with are the fact that it only catches syntactic dependencies and that it relies upon having access to .class files or generated .bsc files to analyze projects.

Concerning the fact that Lattix LDM only catches syntactic dependencies, it would not be as useful in a system with a lot of semantic dependencies. One example of such a system would be a system that uses implicit invocation, such as the A3 project displayed in Figure 3. In this system, components use the event bus to interact with one another; they send events out onto the event bus and also register to receive certain events from the bus. The ability of Lattix LDM to track dependencies stops at the event bus. As Figure 3 depicts, the handlers do not have any syntactic dependencies upon each other, even though there are semantic dependencies between the handlers. It is possible to manually mark the dependencies among the various handlers, so not all is lost; however, this can be a tedious task, especially for large projects.

Overall, it would be difficult to use Lattix LDM in any project that is based upon a lot of indirect communication, such as implicit invocation and shared memory. The semantic dependencies would not be detected automatically by the tool, which could give an inexperienced user or someone not familiar with the system an unclear view of the dependencies in the system.

Another example where only tracking syntactic dependencies creates unusual results deals with polymorphism. We injected polymorphism into the A3 project in the following manner:

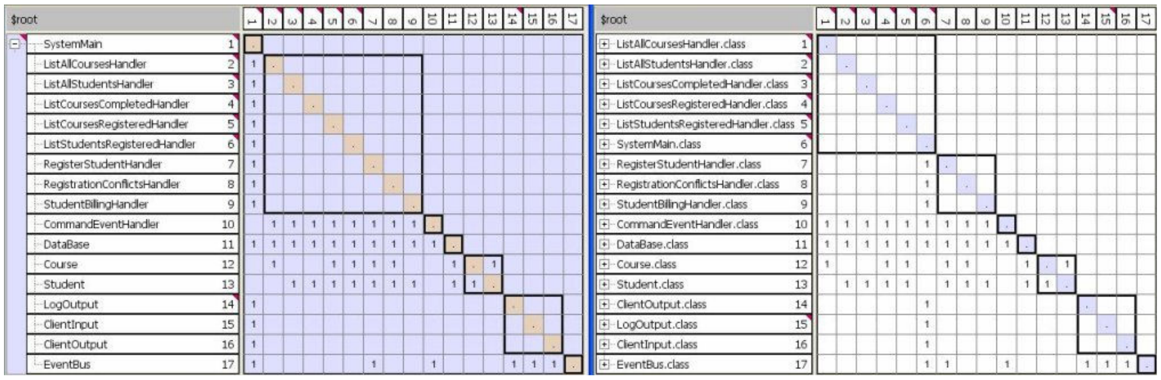
Original Code:

```
ListAllStudentsHandler objCommandEventHandler1 =
                     new ListAllStudentsHandler(
                       db,
                       new
                         int[]{EventBus.EV_LIST_ALL_STUDENTS},
                       EventBus.EV_SHOW);
.
.
.
```

Polymorphism-Injected Code:

```
CommandEventHandler objCommandEventHandler1 =
                     new ListAllStudentsHandler(
                       db,
                       new
                         int[]{EventBus.EV_LIST_ALL_STUDENTS},
                       EventBus.EV_SHOW);
.
.
.
```

While this code is functionally identical and has the same dependencies, due to the syntactic nature of Lattix LDM, the DSM diagrams generated by the tool identify different dependencies, as can be seen in Figure 8 below.



$root		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
SystemMain	1	.																
ListAllCoursesHandler	2	1	.															
ListAllStudentsHandler	3	1		.														
ListCoursesCompletedHandler	4	1			.													
ListCoursesRegisteredHandler	5	1				.												
ListStudentsRegisteredHandler	6	1					.											
RegisterStudentHandler	7	1						.										
RegistrationConflictsHandler	8	1							.									
StudentBillingHandler	9	1								.								
CommandEventHandler	10		1	1	1	1	1	1	1	1	.							
DataBase	11	1	1	1	1	1	1	1	1	1	1	.						
Course	12		1			1	1	1	1			1	.	1				
Student	13			1	1	1	1	1	1	1		1	1	.				
LogOutput	14	1													.			
ClientInput	15	1														.		
ClientOutput	16	1															.	
EventBus	17	1						1			1				1	1	1	.

$root		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
ListAllCoursesHandler.class	1	.																
ListAllStudentsHandler.class	2		.															
ListCoursesCompletedHandler.class	3			.														
ListCoursesRegisteredHandler.class	4				.													
ListStudentsRegisteredHandler.class	5					.												
SystemMain.class	6						.											
RegisterStudentHandler.class	7						1	.										
RegistrationConflictsHandler.class	8						1		.									
StudentBillingHandler.class	9						1			.								
CommandEventHandler.class	10	1	1	1	1	1	1	1	1	1	.							
DataBase.class	11	1	1	1	1	1	1	1	1	1	1	.						
Course.class	12	1			1	1		1	1			1	.	1				
Student.class	13		1	1	1	1		1	1	1		1	1	.				
ClientOutput.class	14						1								.			
LogOutput.class	15						1									.		
ClientInput.class	16						1										.	
EventBus.class	17						1	1			1				1	1	1	.

Figure 8. Semantic Dependencies for the A3 project. (Identical to Figure 3)
(Left) DSM for the original project; (Right) DSM for the project with polymorphism

The dependence on having access to .class files for Java projects and a .bsc file for C/C++ projects can also limit the applicability of Lattix LDM to a project. Projects that use a significant amount of external libraries without having access to the code or .class/.bsc files could be difficult to analyze with the Lattix LDM tool. Also, the Lattix LDM C/C++ help instructions rely upon having Microsoft's Visual Studio to create .bsc files, which requires additional software, and thus, can further limit the applicability of Lattix LDM to a project.

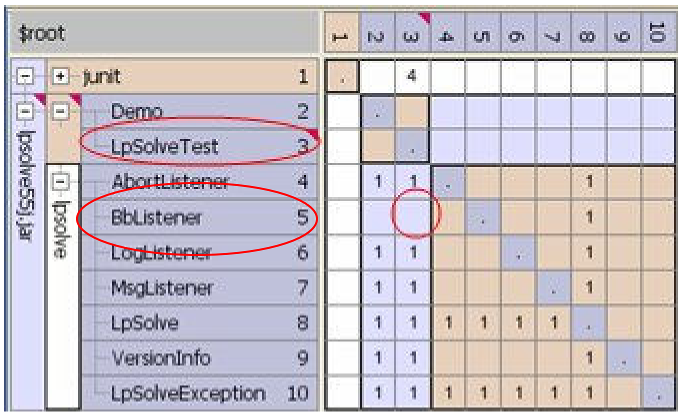
3.2. Strengths

A strength of the Lattix LDM tool is the support that it has for hierarchical structures, such as packages and classes in Java. Having this hierarchical nature displayed with the DSM structure is very useful, as discovering the dependencies among packages can be quite useful without having all of the details concerning the classes cluttering up the diagram. Being able to analyze dependencies at various levels within the project is very useful as certain information can be abstracted away to focus on pertinent portions of the system. For example, a developer of a certain class can check the dependencies of his class file in order to ensure that it has no dependencies on certain packages.

Another good aspect of the tool is the use of the DSM partitioning algorithm that the tool utilizes to reorganize the diagram. We used this to organize the diagram into logical subsystems based upon their dependencies. Qualitatively, the algorithm seems quite good and the subsystems that it identified seemed reasonable. Using the partitioning algorithm is also quite useful as it transforms the diagram into a more familiar form in which certain architectural styles or deviations can rapidly be identified; this is especially useful in identifying how well the system conforms to the layered style.

The speed of the tool is also another strength. Loading a project into Lattix does not take a significant amount of time in our experience. During our evaluation, even the largest projects took less than thirty seconds to be loaded into the tool. However, the help documentation for C/C++ use mentions that .bsc files take longer to load into the tool than do Java .class files. Specifically, the documentation says that it can take 2+ hours for a system with 10000-20000 files to be loaded into the tool. The partitioning algorithms also execute quite rapidly. The speed of the partitioning algorithm is particularly pleasing because an architect that is going through and making changes to dependencies can quickly see the reorganization of identified subsystems as incremental changes are made to the project and the corresponding DSM.

A somewhat tangential use of the tool is that it can also be used to identify holes in code coverage for a test suite. We noticed this with the lpsolve project, as, within a set of unit tests, there was a class with a missing dependency, which indicated that the class might not be tested. This is shown in Figure 9. Thus, using this tool could be a good way to get a quick overview of how well a test suite covers the classes within a project.



		1	2	3	4	5	6	7	8	9	10
junit	1	.		4							
Demo	2		.								
LpSolveTest	3			.							
AbortListener	4		1	1	.				1		
BbListener	5					.			1		
LogListener	6		1	1			.		1		
MsgListener	7		1	1				.	1		
LpSolve	8		1	1	1	1	1	1	.		
VersionInfo	9		1	1					1	.	
LpSolveException	10		1	1	1	1	1	1	1		.

Figure 9. Possible hole in lpsolve unit test class.

3.3. Weaknesses

The primary weakness that is present in the Lattix LDM tool is the lack of support for semantic dependencies. While it is understandable that this is a much more difficult analysis problem, it is a significant blind spot in the tool. As described in the Scope section of this report, the weakness in discovering dependencies in an implicit invocation system; a system containing polymorphism; or in any other system where indirect communication methods are used can easily lead to a misunderstanding of the system, even for an experienced user. However, to the tool's credit, it explicitly states in the OOPSLA [1] paper that the tool only identifies syntactic dependencies in a system.

Another issue dealing with semantic dependencies is transitive dependencies. An example would be that A depends on B, and B depends on C. A indirectly depends on C, but this dependency does not show up in the diagram, which is understandable. (Having all transitive dependencies displayed on the diagram would likely lead to a really messy diagram in which many components would seem to depend on many other components.) It's possible to trace the transitive dependencies by hand on the diagram, but this can be quite tedious. It would be nice to have an option to select a single row on the diagram and to then show all of the "dependency chains" that it are involved with that row. It's likely that this kind of information might be easier to understand in a more traditional box-and-line-type architectural description.

An annoyance and scaling issue with the tool is that the DSM partitioning algorithm can only be applied to a single package at a time. This means that the DSM partitioning algorithm cannot be applied to the entire DSM at once, which can lead to a lot of tedious work in a large project.

While the tool claims to support C/C++ projects, there is a significant dependence on outside tools to generate the .bsc files that are necessary to run the Lattix LDM tool on such projects. The help file details using Microsoft's Visual Studio to create the .bsc file, which may not be possible for some users. Having some other way to generate the .bsc files within the tool would be quite useful as it would allow the tool to be used with any C/C++ project, instead of just those created with Microsoft's tools.

Concerning box-and-line-type architectural descriptions, Lattix LDM provides an option to generate an architectural diagram. This part of the tool is not ready for heavy use. The diagram generated for the ParkNPark project (Figure 10) is displayed below. It is quite a complicated architectural diagram that is made worse by the fact that the fonts seem way too large for each of the boxes. Expanding the boxes with the "+" sign also causes some boxes on the diagram to start overlapping other boxes, further obscuring the diagram. In the tutorial portion of the tool's "Help" section, it says of the conceptual architecture diagram that "the positioning is suggestive of dependencies and conforms to our intuition." Further along in the tutorial, it indicates that the horizontal splits indicate layering and the vertical splits indicate independence of components from each other. However, this is not very clear from the figure. This portion of the tool does not seem as useful as the DSM-centric portions.

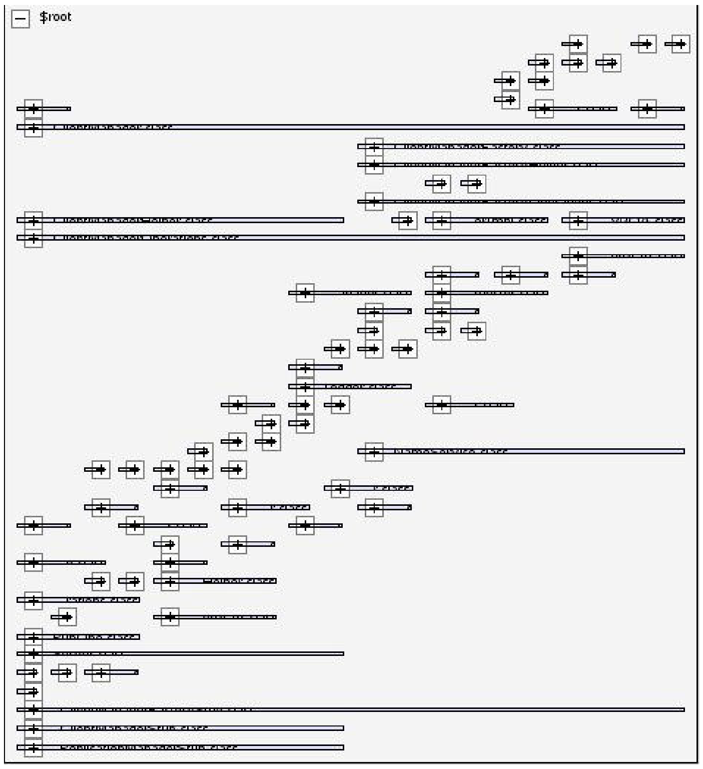


Figure 10. Tool-generated Architecture for the ParkNPark Project

4. Benefits

Using this tool in a software project has many benefits. One of the main benefits is that of scalability. In a real-world software project environment, projects can get very large, very complex, very quickly. Having a tool that can scale well with the size of the project is very beneficial in this case. For example, if a project team loses track of the intra-project dependencies because the software project is growing at a rapid pace, the Lattix LDM tool can be used to obtain the dependency information. Additionally, since it is based on the DSM approach, even if the project gets very large, the dependencies can still be displayed in a very clear and clean manner. On the other hand, if box-and-line diagrams were used to represent dependencies, then, as the project became larger, these diagrams would turn into a chaotic mess that would be very difficult to comprehend and analyze.

The tool also provides the benefit of allowing users to quickly analyze dependencies at different levels of the project. Since the tool supports hierarchical structures in DSMs, it allows users to quickly analyze dependencies between packages. If a user then wants to delve deeper to figure out why a dependency exists, then she can expand the package structure to see what components within the package are responsible for that dependency. This abstraction is very conducive to the user's efforts because it doesn't plague her with details that she doesn't care about. For example, if she wants to deploy the software project, then she can look at the dependencies between packages in order to determine how best to deploy the packages over a distributed environment; if she cares about performance, then packages that have a lot of dependencies on one another would be deployed on machines that are physically close to one another, and packages that don't have a lot of dependencies on one another could be deployed on machines that are geographically distant.

Another benefit that the tool provides is that of architectural discovery. The DSM could be used to determine how well the software project supports certain architectural styles. The partitioning algorithm that Lattix LDM provides greatly helps with this process. After applying the algorithm, the DSM is rearranged such that it is as close to a block-triangular structure as possible. For example, if the algorithm succeeds in forming such a structure, then the users know that the software project supports a layered-like architectural style. Additionally, if the dependencies appear such that a row only depends on the row immediately above it, then it shows that the project supports a pure layered style. The DSM could also be used to decide how best to group components into modules so as to reduce the number of dependencies between the components; this can help make the system more modular, and thus, more modifiable and maintainable.

5. Application to Practicum

Having the Community Version of this tool is quite handy as it will allow us to use it on our practicum project. The Decision Support System for Efficient Aid Distribution (DSS4EAD) is further described at [3], but the primary purpose of the system is to allow the centralized collection of information about communities and to then use that information to determine the most efficient use of aid funds to maximize social benefit. We are implementing this system with quite a few open-source components, and one of the main architectural drivers is to keep the system modifiable for future developers.

It will be possible to use the Lattix LDM tool as we will be implementing the system in Java. Using the tool will allow us to track dependencies to ensure that the components that we create do not have multiple dependencies upon the open-source components that we use. This will allow future developers to have clean breakpoints within the architecture where they can remove certain open-source components and insert their own or other commercial components; this will allow them to achieve certain quality attributes, such as performance. Having a clean layering implementation and avoiding cyclical dependencies, except where absolutely necessary, will allow us to deliver a modifiable system to our customer.

Because of these benefits, we do plan to use Lattix LDM on our practicum project. More specifically, we will use the DSM functionality of the tool. We will use the tool to compute the DSM for our project at every milestone in order to ensure that the implementation is in accordance with the design considerations. We may also use it in between milestones for the purpose of sanity checks. It may also be useful when determining how design tradeoffs affect the architecture (e.g. we don’t want to introduce dependencies that will break conformance to the architectural style). Additionally, as the Conceptual Architecture functionality of the tool did not seem to be very helpful to us, we do not plan to use that functionality.

6. Conclusion

Overall we were quite pleased with the tool. While the lack of support for semantic dependencies is disappointing, it is not fatal for this tool as manual input from knowledgeable architects and developers can cover up this blind spot. The Lattix LDM tool seems like a very good place to start for architectural discovery as the syntactic dependencies and DSM partitioning algorithm can help to quickly understand the logical subsystems of a system, even in large projects.

The quality of the tool is quite good as well, as should be the case for a commercial product. There were no glaring bugs in the primary DSM functionality and only a few annoying user interface issues. The conceptual architecture diagram could certainly use some work, but this almost seems like a portion of the tool that could be left out without significantly sacrificing the usefulness of the tool.

Using Lattix LDM to track dependencies in a project seems quite doable and very useful given the amount of time and effort involved. It seems especially useful if you can avoid projects with a lot of semantic dependencies. Lattix LDM seems to favor Java projects much more then C/C++ projects, but C/C++ projects would still be possible if you are already using Microsoft tools as part of your development method.

Another positive note for Lattix LDM is that is uses a technique that has already been successfully used in other industries. The DSM technology has had a chance to mature already before its application to software. Being based upon the DSM technology will also allow Lattix LDM to take advantage of new, clever algorithms that may be invented in the future; these algorithms may improve on the current algorithms by partitioning the DSM diagrams in different ways in order to reveal other aspects of complex systems.

References

[1] Sangal, Neeraj, Ev Jordan, Vineet Sinha, Daniel Jackson. *Using Dependency Models to Manage Complex Software Architecture*. ACM. October 2005. http://www.lattix.com/download/dl/oopsla05.pdf

[2] More information about the Eclipse IDE can be obtained from the Eclipse web site at http://www.eclipse.org

[3] More information about the Decision Support System for Efficient Aid Distribution (DSS4EAD) can be found at http://heinz-aiddist.heinz.cmu.edu