Repository System for Geospatial Software Development and Integration

Basem Y Alkazemi
Department of Computer Science
Umm Al-Qura University (UQU)
Makkah, Saudi Arabia

Abstract—**The integration of geospatial software components has recently received considerable attention due to the need for rapid growth of GIS application and development environments. However, finding appropriate source code components that can be incorporated into a system under development requires considerable verification to ensure the source code can work correctly. This paper therefore describes the design of a repository system that employs a new specification language, namely SpecJ2, to address the challenges involved in integrating and operating software components. SpecJ2 was designed to represent the architectural attributes of source code components and to abstract their complexity by applying the notion of *separation of concerns*, a key consideration when designing software systems. The results of the experiment showed that SpecJ2 is capable of defining the different architectural attributes of source code components and can facilitate their integration and interaction at run-time. Thus, SpecJ2 can classify software components according to their identified types.**

Keywords—Open-Source software; geographic information system; repository system; specification language; components integration

I. Introduction

There are many open-source GIS projects now actively running and most have reached a high level of maturity in applying their tools to the provision of information that can feed into decision-making processes [1]. GIS applications have evolved rapidly by integrating different components to generate a fully functional system that serves a specific domain [2]. Business requirements are the key driver in defining the architecture of any GIS application in terms of identifying the functional components related to: data collection and remote sensing components; storage and retrieval components; semantic analyses and data geoprocessing components; and presentation and reporting components. Moreover, certain GIS applications might need to be integrated as a whole into different types of systems to address certain performance, usability, and reliability issues. Despite the functional advantages of open-source GIS-component integration, ensuring the interoperability of different components is a very challenging task. In technical terms, a comprehensive environment is required to define the necessary integration frameworks and avoid potential mismatches between GIS components, both syntactically and semantically [3]. Moreover, the diversity of available OSS-GIS solutions might confuse normal users and complicate the process of identifying the best GIS tool for users in terms of the functionality, usability, and integration of applications with other platforms. This paper therefore aims to establish a general-purpose repository system that identifies, classifies, integrates, and develops open-source GIS components to fulfill the requirements of GIS business applications. Specifically, the paper addresses the difficulties involved in component integration as this is the key element underpinning the development of GIS applications. The terms “source code components” and “software components” will be used interchangeably throughout the paper as both refer to source code fragments.

II. Related Work

The integration of components has been a research topic in different application domains from early work by Allen et al. [4] through to the present day, where further investigations into components or services integration continue to be reported.

For instance, Suri et al. [5] examined modularity and interoperability aspects for software systems in industry from an integration perspective. They discriminated between source code behavior and the execution logic within the systems. They utilized UML to bridge the gap between behavioral modelling and the execution of systems. Kaur and Singh [6] developed a web service called GlueCode to mediate the interaction between components written in different programming languages, such as java-based components and .Net components, and the data source Cloud.IO. Their primary focus was on the data exchange patterns and signature matching between components. Farcas et al. [7] developed a new real-time component model to address the problem of component integration. They identified the key distinguishing factors of software components that need to be addressed to ensure successful integration, such as component behavior and a logical execution environment. Fatima et al. [8] conducted a semi-systematic survey to identify risk factors for the integration of software components. They concluded that a lack of interoperability standards, glue code, and format variation are the key reasons for failure to integrate. Schorp and Sommer [9] defined a new component model in the domain of automotive ICT architecture. They contended that a successful integration of software components can be accomplished if functional interdependencies and non-functional requirements are clearly addressed. Their component model facilitated integration based on the discovery of interaction between features. Dogra et al. [10] investigated the reasons for component integration failure and concluded that such failure is primarily attributable to architectural mismatches between software components. Furthermore, they highlighted the fact that a lack of knowledge and expertise regarding software components might also cause problems with integration.

Overall, most of the reported work has thus identified component architecture as the key hindrance to successful integration. There have been few studies showing that functional interdependencies might also case integration failure which means this area of research requires further investigation. This work proposes a new methodology to document and facilitate component interaction by considering the architectural attributes of source code components. It reports our ongoing development of a software development environment that facilitates the identification and integration of software components to build a GIS functional application.

III. Repository System Design for OSS-GIS Tools

A repository system is a development environment that is equipped with the necessary tools for the automatic identification, classification, and storage of software components. Users can retrieve components from the repository in accordance with their functional requirements by conducting a free-text search, browsing, or providing a detailed formal system specification. In this section, we describe our proposed repository solution for open-source GIS software systems. We also explain the main architecture of the repository system. The main objectives when designing this repository system were to:

1) Establish foundations for open-source software within organizations to support internally run projects
2) Assist in identifying appropriate open-source tools for projects
3) Eliminate the licensing costs associated with proprietary software
4) Address the lack of support that hinders many organizations with respect to utilizing open-source GIS software systems
5) Provide the necessary awareness and educational support for open-source GIS software systems
6) Collaborate with different colleges and universities to embed open-source GIS tools into their course plans.

As illustrated in Fig. 1, the system developed through this work contains the following five key sub-systems:

- Components Identifier
- Classifier
- Builder
- Meta-data store
- Matcher

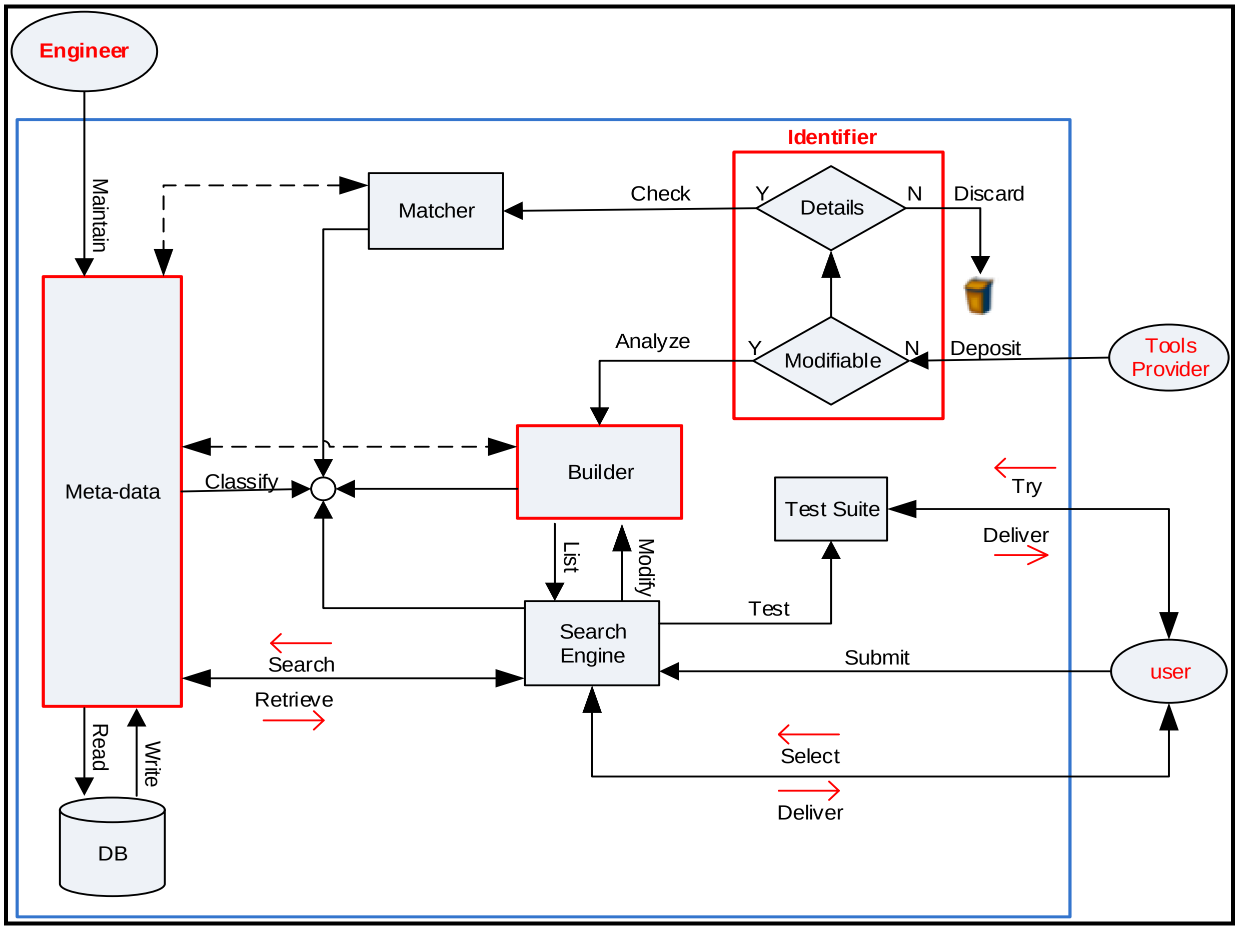

Fig 1. GIS Repository System Architecture.

The behavior of the repository system is described as follows. The source code of a GIS component is deposited into the repository system either manually by uploading code to the system or by providing a GitHub URL from which to import the source code. Once the source code is uploaded into the repository, the identifier sub-system analyzes the code to identify its architecture. Based on this analysis the component might be classified under a matching category represented in the classifier sub-system. If the source code cannot be categorized under any of the available categories it is discarded from the repository workflow and stored as an "Undefined Type" in the repository for further consideration.

From a user's perspective, the repository system provides the capability to search for an available source code or sub-systems by providing an XML description of component types using the developed specification language described in Section 5. The matcher sub-system compiles the XML description provided by the user to identify a match to the components in the repository. Matching specifications result in finding either exact matches to the description or partial matches. If exact matching components are found, they are listed to users for further investigation. If partially matching components are found, the repository system refactors the source code to fulfill the XML description that was provided. In cases where the available source code in the repository lacks some of the required interfaces to match the user's specification, the repository generates the necessary interfaces in the form of skeleton code to satisfy these requirements. However, the code generated by the refactoring process must be examined by the user to confirm that the new packaged component works and will provide the expected behavior.

IV. Repository Classification Scheme

The GIS system architecture, like many information systems, commonly conforms to the N-Tier architecture [11], which is characterized by three main layers: the interaction and presentation layer, the processing layer, and the management layer. The overall architecture is depicted in Fig. 2.

These three layers are the building blocks of many GIS systems, whether they are proprietary GIS software systems or open-source GIS software systems. Our classification scheme was primarily built on these layers to identify high-level functional areas and their facets for the classification of GIS tools. It is necessary to understand these layers and define their interfaces in order to facilitate the potential integration of different components, such as those found in other GIS tools.

As highlighted by Dempsey [12], many OSS-GIS tools are available to support these three layers. For example, according to Alkazemi et al. [13], in the information management layer, common tools include PostGIS and Geodatabase, both of which serve as a data source and database for other tools. PostGIS and Geodatabase make it possible to store GIS data in a central location for easy access and management. Grass, Sextante, and MapWindow are some of the common tools used for the human interaction layer; these facilitate communication between the information system and external users, which are either people or computer systems such as a web browser. Hadoop [14] is one of the OSS tools available on the market and is classified under the processing layer. Ut is an Apache top-level project that is being built and used by a global community of contributors and users.

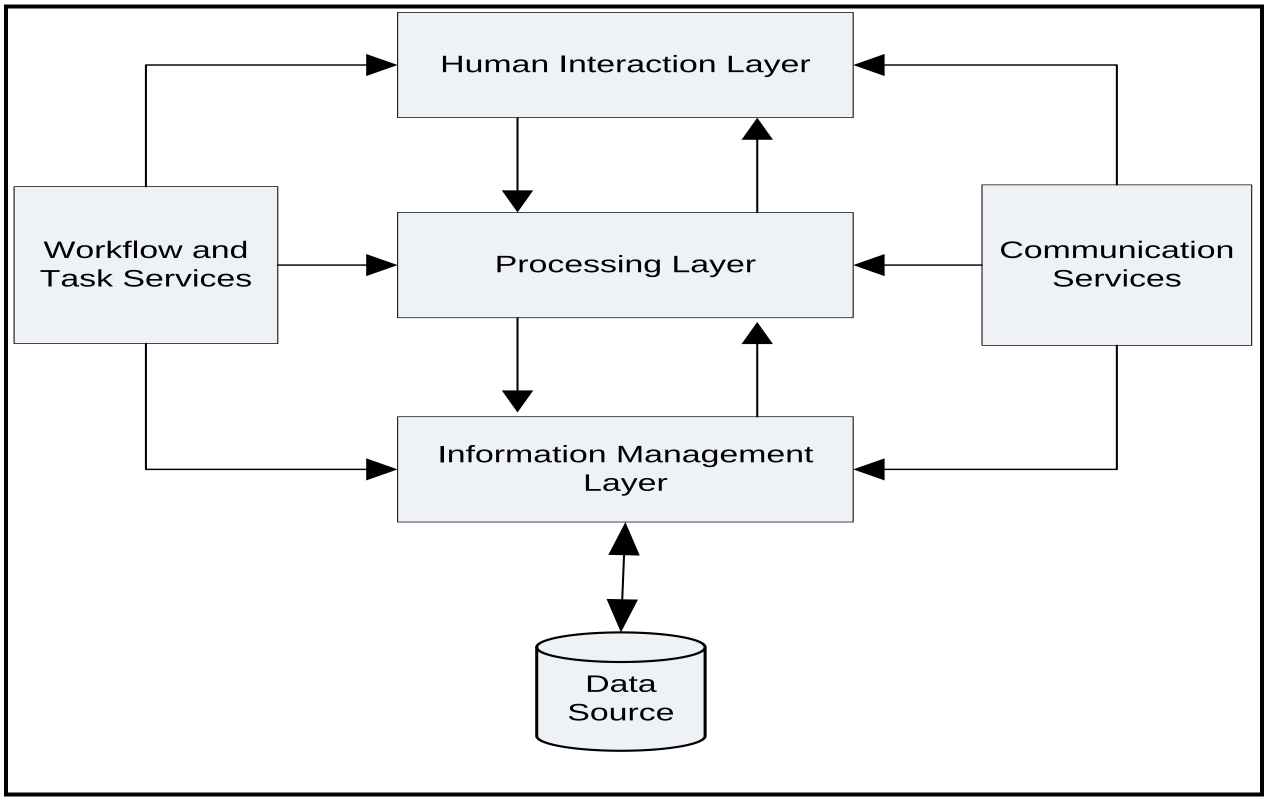

Fig 2. N-Tier GIS Application Architecture.

V. Integration of GIS Components

Software components can interact with each other as services if they share common characteristics as a data exchange model [15]. However, to work correctly, source code components must comply with standard characteristics. Thus, source code components might be characterized by:

- Signature
- Programming language
- Behavior
- Sequence of execution
- Dependencies

Signature of methods or functions defines the name of the method, input and output parameters, and their datatypes. Programming language adds more filtration to the searching text to obtain a more accurate result. Certain source codes may not be used alone and can be incorporated with other codes or applications. Therefore, it is necessary to understand the sequence with which a method is executed to run as expected in the application under development. The attributes of source code components, especially those related to their architectural attributes, are always hard to document and represent as they differ from one programming style to another.

To avoid the complexity of source code matching characteristics, we developed a specification language, namely SpecJ2, to summarize and document the necessary attributes of source code components. SpecJ2 formalizes some of the architectural characteristics of software components and this also applies to GIS-component integration. SpecJ2 thus serves as a verification mechanism that checks whether source-code conforms to the required properties of a system in the OSS-GIS repository system. Table 1 describes the syntax of the SpecJ2 language that identifies the key elements which represent the architectural properties of components. Some of the attributes may be null values and therefore might be omitted in the description file. The key attributes are data input and output as these handle data exchange between the components of the system. Thus, SpecJ2 can be considered the adapter layer between any two GIS components designed to interoperate with each other as it handles component interoperability. Thus, data are exchanged in a standard manner between the different types of components. This layer is generated automatically by the builder component within our repository system to facilitate the simultaneous integration of tools or components. The conceptual view of SpecJ2 is presented in Fig. 3.

SpecJ2 represents the intermediate layer (i.e. wrapper) between source code components and the underlying framework of the system to be built. It hides the complexity of the implementation and differences in software components within the framework. Thus, if a developer compiles the system under development all the components will be considered the same because the SpecJ2 layer hides component types from the underlying system compiler. Furthermore, SpecJ2 defines the linkage between components that will exchange messages by connecting the interfaces of methods together, which facilitates data exchange at run-time. For example, if the system under development was built using Java language and a developer needed to incorporate a component written in another programming language, say PHP class, they can either treat them as services and handle data exchange at run-time or use SpecJ2 to handle environmental difference parameters.

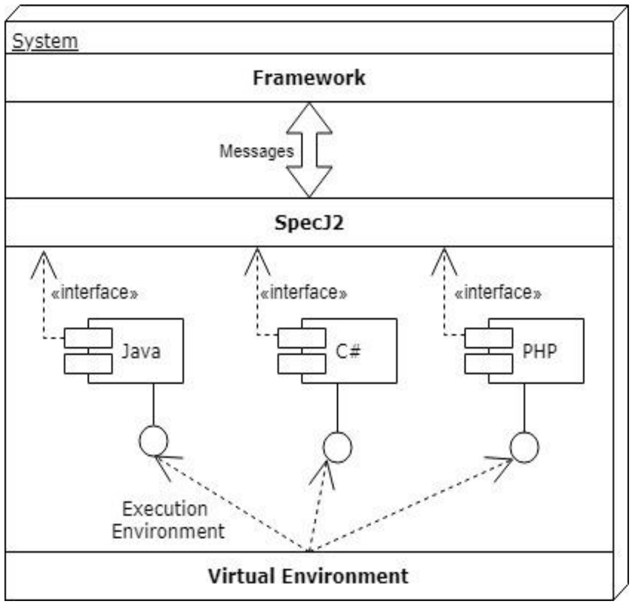

Fig 3. SpecJ2 Conceptual View.

TABLE I. SpecJ2 Syntax

Tag	Description
<SpecJ2>	Identify a document under SpecJ2 specification
<SpecJ2>\<name>	Define the name of the type
<API>	Capture the architectural attribute of the component type
<API>\< Code_Scope >	
<Code_Scope>\<name>	Define memory name
<Code_Scope>\< Input_Stream >	Define component input data stream
<Code_Scope>\< Output_Stream >	Define component output data stream
< Code_Scope >\<Failure>	Define exception handling mechanism
< Code_Scope >\<File>	Define external file that architectural type use to operate
< Code_Scope >\<Storage>	Define cache memory
<Input_Stream>\<sequence>	Identify sequence of input data
<Output_Stream>\<sequence>	Identify sequence of output data
<Order>\<type>	Define data type
<Failure_Handling>\<type>	Define type of exception handling
<Perquisites>\<lib>	Define required resources
<File>\<name>	Define name of file
<File>\<type>	Define type of file
<Memory>\<name>	Define memory address
<Memory>\<type>	Define memory type
<File_type>\<sub-type>	Define specialized generic file type

VI. Experimental Setup

In SpecJ2 we described the geocoding module of ArcGrid which is a generic functional model in many forms of geospatial software as it interprets coordinates (i.e. latitude, longitude) based on their corresponding addresses, either by querying the database of stored addresses (e.g. Google API) or by reading addresses from points on the map. To demonstrate our approach, in Fig. 4 we provide a description of the logging component of the geocoding facility in SpecJ2.

```
<SpecJ2>
    <name>
        ArcGeo_Logger
    </name>
    <API name='Logging'>
        <Code_Scope identifier='getLogger'>
            <input_stream>
                <sequence>
                    <Parm_Type>String</Parm_Type>
                </sequence>
            </input_stream>
            <output_stream>
                <Parm_Type>Logger</Parm_Type>
            </output_stream>
        </Code_Scope>
    </API>

    <API name='config'>
        <Code_Scope identifier='config'>
            <input_stream>
                <sequence>
                    <Parm_Type>String</Parm_Type>
                </sequence>
            </input_stream>
            <output_stream>
                <Parm_Type>void</Parm_Type>
            </output_stream>


        </Code_Scope>
    </API>

    <API>
        <Code_Scope identifier ='ArcGridReader'>
            <input_stream>
                <sequence>
                    <Parm_Type>Object</Parm_Type>
                </sequence>
                <output_stream>void</output_stream>
                <failure_handling>
                    <type>DataSourceException</type>
                </failure_handling>
            </input_stream>

        </Code_Scope>
    </API>
    <prequisit>
        <lib>java.util.logging</lib>
        <lib>org.opengis.referencing.FactoryException</lib>
        <lib>java.io.File</lib>
    </prequisit>
</SpecJ2>
```

Fig 4. Logger SpecJ2 Description.

The SpecJ2 description captures part of the logging capability which is a generic feature in many GIS applications. We conducted our experimental work at this stage by identifying how many components obtained from open source repositories can fit as a logging module, and hence can be reused in GIS applications. We therefore obtained 50 codes for each component type from GitHub; these were defined as geospatial related components from solutions including uDig, ArcGrid, and deegree [12]. However, we limited the experiment to Java based solutions. The selection of the source code was carried out manually by downloading all the corresponding JAR files of the solutions then applying the sampling technique defined by Kamal et al. [16] to ensure we covered as many of the test samples as possible. We then ran SpecJ2-compiler to scan through the source code to identify matching results. The process of compiling source code is illustrated in Fig. 5.

Source code is first examined using the extraction tool that identifies the signature of the methods within the JAR file provided. The extracted methods are then sent to the SpecJ2-compiler to compile the source code against a generated Junit test class based on the XML component description provided. Fig. 6 presents the generated JUnit test class used for compiling test samples. In cases where the deposited source code does not match any component types, re-scoping of the source code fragment was performed to include more attributes for the next round. Re-scoping was initially set for four rounds. If components failed to compile after the first round they were discarded from the system.

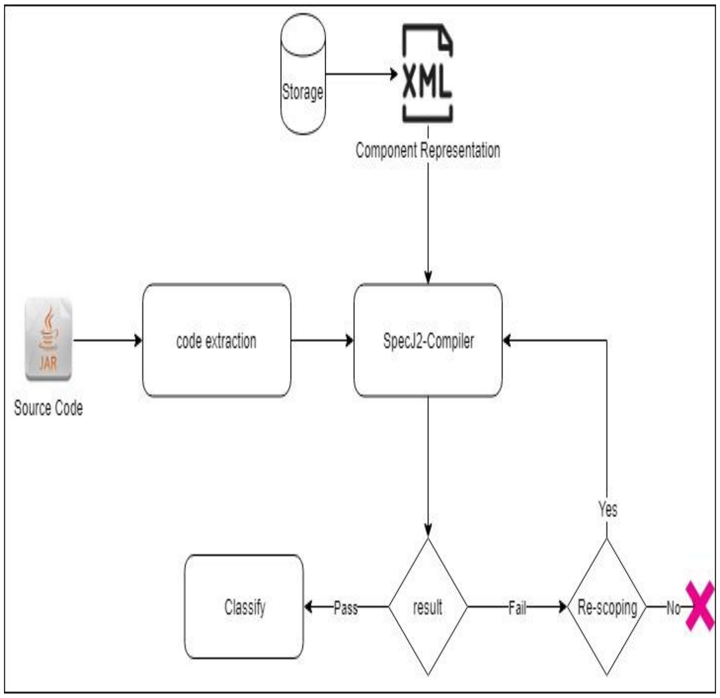

Fig 5. SpecJ2 Operation in the Repository.

```
import static org.junit.Assert.*;
import org.junit.After;
import org.junit.Before;
import org.junit.Test;
import specJ2.Compiler.*;
import specJ2.Identifier.*;

public class SpecJ2_Logger_Test
{

    int match_count = 0;
    String [] methods ;
    String [] methods_s;
    Results r = new Results();
        @Before
    public void setUp(Object Target, Object<SpecJ> Source)
    {
        // prepare the component and examin the dependencies
        methods = Target.getMethods();
        methods_s = Source.getMethods();
    }

        @After
    public void tearDown()
    {
        String fileContent = r.getResults();
        FileWriter fileWriter = new FileWriter("c:/SpecJ2
_comiler/Test_Logger.txt");
        PrintWriter printWriter = new PrintWriter(fileWriter);
        printWriter.print(fileContent);
        printWriter.close();
            }

    @Test
    public void SpecJ2_Compile()
    {
        for (int i = 1 ; i < methods_s.length; i++)
        {
            for (int j = 1; j < methods.length; j++)
            {
                if (methods[j].name.matches(methods_s[i].name)
&& methods[j].parms.matches(methods_s[i].parms))
                {

if(methods[j].rtnType.matches(methods_s[i].rtnType))
                    {
                        match_count++; // identfied match
                    }
                }
            }
             if (match_count == methods_s[j].length)
             {
                 r = run(methods[j]); // compile the code to tag
it as pass or fail.
             }
        }

    }
}
```

Fig 6. SpecJ2 JUnit Test.

VII. Results and Discussion

The results obtained for the experiment are summarized in Table 2. We categorized these results into fully matched, partially matched, and no match. Fully matched refers to when all the attributes defined by the source code component matched the corresponding SpecJ2 description, hence the component can be used without any modifications. However, if none of the attributes were identified in the selected source code, the code fragment is categorized as no match. Midway between both extremes are partially matched components which require further investigation. We counted the number of matching and non-matching attributes to assess the level of modification needed.

TABLE II. Experimental Results

Type	Number of Samples	Fully Matched	Partially Matched			No Match
			Total	Matched Attributes %	Unmatched Attributes %	
vGid	50	26	15	83%	17%	9
deegree	50	39	7	51%	49%	4
ArcGrid	50	43	6	88%	12%	1
OpenJUMP	50	37	13	42%	58%	0
QGIS	50	44	6	61%	39%	0
gvSIG	50	33	8	39%	61%	9

The experiment produced striking outcomes with respect to the identification of component types. Overall, SpecJ2 yielded significant results in terms of matching components to the types defined in the repository. Compared to the matched samples, the number of unmatched components was minimal with an overall average ratio of 0.124 (i.e. for each "no match" there was four matched components on average). We therefore conclude that SpecJ2 is useful in representing source code components and can also be used to intermediate the interaction between various types of component. The results of the partially matched components were twofold as the overall percentage of matched attributes counted was more significant than the percentage of unmatched attributes except in the cases of openJUMP and gvSIG. We investigated the source code for these component types by hand and observed that openJUMP needed to operate in conjunction with the OSGE framework to provide a complete set of attributes. However, gvSIG was slightly different as the available components were mainly plugins, hence the attributes examined were an extension of the main framework. The other missing attributes were coded in the main Factory class within the gvSIG package. Thus, the unmatched percentages indicated that they were missed by SpecJ2 due to a lack of support for inheritance which will be included in the new release of the language.

VIII. Conclusion and Future Work

The integration of software components is a key element of the component-based software development paradigm. The architectural and the behavioral features represent the backbone of any integration process and must be described precisely. The development of GIS applications is no different as it involves various forms of component integration.

In this work, we developed SpecJ2 as a specification language to address the complex interoperability and execution of software components. SpecJ2 complemented the design of the repository system proposed in this work to examine the feasibility of identifying component types and classifying them according to their attributes. The results obtained in this work supported the design considerations of SpecJ2 and proved that it was capable of identifying potential mismatches between software components. Such identification is significant as it can help developers verify components prior to reusing them in their systems.

The next step in this work is to automate the refactoring mechanism of software components to transform those which are partially matched into fully matched candidates. Moreover, we plan to consider a wider range of component types in different programming languages.

Acknowledgment

The Author of this work would like to express his gratitude to Umm Al-Qura University for supporting the investigation and development of the different models of this work.

References

[1] Steiniger, S. and Weibel, R.. "GIS software: a description in 1000 words". In Encyclopedia of Geography, B. Warf, Ed. London, UK: Sage. (Available on-line at: http://dx.doi.org/10.5167/uzh-41354.), pp. 1-4, 2010.

[2] Neteler, M. and Mitasova, H. Open Source GIS: a GRASS GIS approach (3rd Ed.). New York: Springer. ISBN 978-0-387-35767-6.406, 2008.

[3] Dejan, J. and Radmila, M. "Integration opensource GIS software for improving decision-making in local community", Acta Technica Corvininesis-Bulletin of Engineering, Volume 6 Issue 4, pp. 73-76, 2013.

[4] Allen, R., Garlan, D. and Ivers, J. "Formal Modeling and Analysis of the HLA Component Integration Standard", in Proceedings of ACM SIGSOFT FSE'98, pp. 70-79, 1998.

[5] Suri K., Cuccuru A., Cadavid J., Gerard S., "Gaaloul W. and Tata S. Model-based Development of Modular Complex Systems for Accomplishing System Integration for Industry 4.0". In Proceedings of the 5th International Conference on Model-Driven Engineering and Software Development, pp. 487-495, 2017.

[6] Kaur, M., Singh, P. "Integrate the Components with the Help of Glue Code Using .Net And Java Platform", International Journal of Advanced Research in Computer Engineering & Technology, Volume 4, Issue 4, pp. 1266-1270, 2015.

[7] Farcas, E., Farcas, C., Pree, W. and, Temple, J. "Real-time component integration based on transparent distribution". In Proceedings of the second international workshop on Software engineering for automotive systems (SEAS '05). ACM, 2005.

[8] Fatima, F., S., Ali, M.U. and Ashraf, M.U. "Risk Reduction Activities Identification in Software Component Integration for Component Based Software Development (CBSD)", International Journal of Modern Education and Computer Science, Volume 4, pp. 19-31, 2017.

[9] Schorp, K. and Sommer, S. "Component-Based Modeling and Integration of Automotive Application Architectures", IEEE International Electric Vehicle Conference (IEVC), Florence, Italy. 2014.

[10] Dogra, N., Sharma, A. and Singh, H. "Component integration: a challenge for component-based software development", International Journal of Latest Trends in Engineering and Technology, special issue, pp. 37-40. 2016.

[11] Fowler, M. Patterns of Enterprise Application Architecture. Addison-Wesley, 560pp.,2002.

[12] Dempsey, C., "Open Source GIS and Freeware GIS Applications". (Available on-line at: https://www.gislounge.com/open-source-gis-applications/), 2017.

[13] Alkazemi, B., Naseer, A., Aldoobi, H. "Towards A Repository System for Open-Source GIS Software Components", 5th Open Source GIS Conference - OSGIS, At Nottingham Geospatial Institute, The University of Nottingham, UK, 2014.

[14] Shvachko, K., Kuang, H., Radia, S. and Chansler, R. "The Hadoop Distributed File System", IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST), Incline Village, NV, USA, 2010.

[15] Blay-Fornarino,M., Charfi,A., Emsellem,D., Pinna-Dery,A., and Riveill,M." Software interactions", Journal of Object Technology, Volume 3, Issue 10, pp 161–180, (Available on-line at: http://www.jot.fm/issues/issues 2004 11/article4), 2004.

[16] Zamli, K., Alkazemi, B. and Kendall, G. "A tabu search hyper-heuristic strategy for t-way test suite generation", Applied Soft Computing, Elsevier, Volume 44, pp. 57-74, 2016.