Computer Science • 14 (4) 2013 http://dx.doi.org/10.7494/csci.2013.14.4.577

Rafał Leszko
Kamil Piętak

VERIFICATION MECHANISM FOR LIGHTWEIGHT COMPONENT-BASED ENVIRONMENT BASED ON IOC CONTAINER

Abstract *This paper presents a concept of component verification framework dedicated to a particular lightweight component environment. The starting point of the paper constitutes a discussion about the significance of verification of syntax inconsistencies in software development. Next, the need of verification in service-oriented and component-based systems is presented, and various approaches of verification in existing component environments are explained. The main part of the paper introduces a concept of functional integrity of component-based systems that utilize verification mechanisms which check consistency between components. The proposed solution is built on a fine-grained component environment (close to classes similarly to the Spring Framework) realized in the* AgE *platform. Selected technical aspects of framework design illustrate the considerations of the paper.*

Keywords verification, component-based systems, functional integrity

1. Introduction

In today's software engineering, applications are hardly ever written from scratch. Instead, they reuse code which already exists in the form of libraries, modules, components, services, or any other external system elements. External code can be provided by different vendors and in different versions. Such a variety of system elements creates a situation where they may be incoherent with each other. This incoherence concerns two aspects: the syntax or the semantics. The semantic consistency is related to the Liskov substitution principle [9] and can be verified with the use of unit tests or learning tests for third-party elements [10]. The syntax is often more difficult to verify, especially in dynamically composed systems in which the late binding [3] mechanism is used. The verification methods for syntax inconsistencies are the subject of this paper.

Verification is the process of checking dependencies of system elements before running the system. According to the level of abstraction and the system architecture [6], a system element can be a function, a module, an object, a component, a service, or any other unit of the system. A function invokes other functions, an object communicates with other objects, and a service uses other services. System elements depend on each other. If any dependent element does not exist, the system fails at runtime. In most modern programming languages, an exception is thrown in such cases; e.g., *ClassNotFoundException* or *MethodNotFoundException* in Java, *NameError* or *AttributeError* in Python.

The main problem with runtime failures is that one can never know whether they will occur or not. The failure may appear at any time or only under certain conditions, which makes debugging process extremely difficult and time-consuming. The verification process lines with the „Fail-fast" approach [13] which states that systems should be designed to immediately report any inconsistencies.

Verification can be partially provided by programming languages. In most compiled languages (like C++, Java, C#), the compilation process itself can be considered as the verification of functions and objects, because the compilation fails with error in case of any problems with dependencies. Nevertheless, it is neither recommended nor desirable to compile all of the code before each system execution. A much better solution is to use components and third-party libraries which are already compiled and, thus, avoid the situation of waiting for the compilation process after each change in the source code.

High-level system elements can be verified by frameworks. This paper discusses the need of verification in Service-Oriented and Component-Based systems and introduces an implementation of the verification system designed for the AgE component platform.

2. Verification in service-oriented systems

Service [4] is a software element which provides a reusable functionality through a well-defined interface. It conceals implementation details insignificant to a client, and exposes an interface independent from the platform and programming language. Due to this independence requirement, such a service interface should be defined in an abstract manner, and the service itself should be treated as a black box which receives the input values and returns the output. Therefore, the service interface can be defined in an independent language; for example, as an XML file (WSDL[1]) [4] in the case of Web Services.

A service can use the functionality provided by another service, known as service dependency. A service is always dependent on some other service's interface, never the implementation. System architecture that is based on many coupled services is called SOA (Service-Oriented Architecture).

SOA is based on the paradigm *find-bind-execute* [1], which is presented in Figure 1. The service provider registers its service in a public registry (e.g., UDDI[2]). Then, a client uses the same registry to find a service, obtain its address, and discover its interface description.

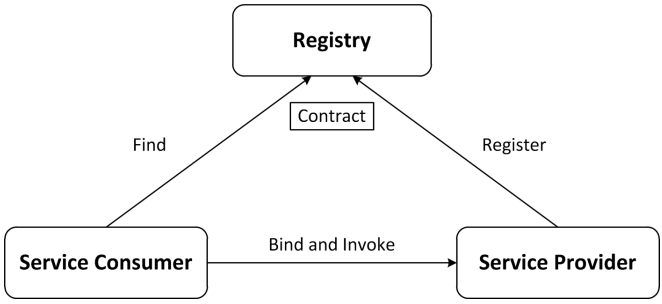

Figure 1. *Find-bind-execute* paradigm.

Service dependency is always known at runtime, never earlier. In most cases, a service dependency is nothing more than criteria by which dependency can by found with the use of the *find-bind-execute* registry. This means that service dependencies are found and bound during execution time, and the availability of other services as well as the registry state may change in time. That is why service-oriented systems cannot be verified a priori.

[1]WSDL (Web Services Description Language) – an XML-based language that is used for describing the functionality offered by a Web service.

[2]UDDI (Universal Description Discovery and Integration) – XML-based registry, by which businesses worldwide can list themselves on the Internet.

3. Verification in component-based systems

A component, according to the classical definition [14], is a reusable entity of composition which has a well-defined interface and context of use. It has no visible state and can be used by third parties.

The definition itself seems similar to the service definition. However, Service-Oriented Architecture (SOA) and Component-Based Architecture (CBA) form different systems. This difference can be observed in the granularity of system elements, the Business-Oriented approach and the programming abstraction level. Nevertheless, the real difference lies not so much in the system itself but in the approach to its creation.

Working with components means working with code and interfaces created for the component environment. In SOA, most important is the functionality provided under the form of a contract. It is not crucial to know the service implementation. Moreover, this implementation may not even be physically in the system, as it may be provided through the network by third-party servers. The necessary knowledge is only: the contract and the information how to find the service (e.g. using *find-bind-execute* registry).

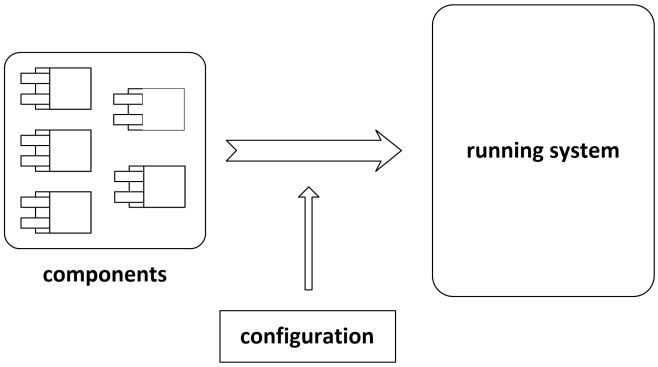

Figure 2. Start-up of a component-based system.

Components are different. While executing a component system, the information about the used components and their availability is known. The system is „composed", which means that all necessary components are gathered together and, by using a start-up configuration, the execution is started (as shown in the Figure 2). Such a system can be verified a priori, because all necessary knowledge is known when the system is launched.

4. OSGi as a hybrid of services and components

OSGi (Open Services Gateway initiative) [7] is a module system and service platform for the Java programming language. It combines the idea of components and services

and, therefore, is a good example for discussing the verification process in different architectural approaches.

An OSGi-based system consists of many modules (called bundles) which can be regarded as components. Each bundle declares its contract in the *MANIFEST.MF* file by describing the exported and imported Java packages. In other words, it declares what a bundle requires and what a bundle provides. By installing appropriate bundles together in the OSGi container, the system is built. This approach is consistent with the Component-Based Architecture — components are gathered together and then the system is started.

The other aspect of the OSGi platform concerns services. Each bundle can register a service and use services registered by other bundles. A service is simply a POJO[3] specially registered in the OSGi context, and its contract is nothing more than a Java interface. OSGi services form a different layer of dependencies and, thus, create another aspect of verification. The essential fact is that services are registered and used at runtime, so their availability is unknown until they are used. That is the main problem with the verification of OSGi services, and that is why the dependencies of services are not being verified.

5. Verification in existing component environments

The OSGi specification introduces a verification process which checks whether a specific bundle can be resolved before started or used by other bundles. This assures that a bundle can be used only if all of its requirements are fulfilled. During this process, the OSGi framework checks if all imported packages exist in the environment and bundles required by the bundle being currently verified. If the required bundles (or those providing imported packages) are not yet resolved, they also have to be verified.

Before any class from a bundle can be used in a system, the bundle has to be successfully verified. Such an assumption allows us to verify all inconsistencies at system start-up, significantly improving system resistance to any changes in the available components set.

Another approach is introduced in Enterprise Java Beans (provided as a part of the Java Enterprise Edition). EJB defines components called beans as classes with additional annotations that specify component type, boundaries, life-cycle methods, and more. Boundary conditions are satisfied at runtime by services of the component environment while instantiating particular beans. Annotations dedicated to validation purposes have also been introduced (defined in JSR 303 [2]). They, however, do not verify component consistency, only the state of particular beans (i.e. they introduce constraints on bean's attributes such as not-null values or size of strings).

Similar solutions exist in Spring Framework, which utilizes an IoC container to produce objects (called beans) based on the start-up configuration (given for example in an XML file). In this case, classes instantiated by the container can be also

[3]POJO (Plain Old Java Object) – an ordinary Java object

perceived components – they define a provided interface and dependencies to other classes expressed by dedicated annotations (e.g. defined in JSR-330 [8]) or convention (e.g. bean convention). The framework can validate in run-time if a given configuration (application context) is proper. This verification process is performed optionally before accessing any bean instance.

The above examples show that the verification process is an important issue in the context of building component-based software. This process assures that systems are properly assembled (i.e. all components requirements are fulfilled). In this paper, the authors present a concept of a functional integrity [12] of component-based systems that utilize verfication mechanisms which check consistency between components. The proposed solution is built on a fine-grained component environment (close to classes similarly to the Spring Framework) implemented in *AgE* platform [11].

6. The realization of Component Dependencies Verification

Component descriptors and a configuration comprised of component definitions are necessary for the verification process, as they constitute an input for the process. The architecture for the verification process consists of *verifier*, verification result, and a collection of independent verification modules which can be attached arbitrarily.

Verifier implements a *Visitor* design pattern [5] and traverses every component definition registered in the IoC Container. While visiting a component, it verifies dependencies according to the proper component descriptor. Any errors are aggregated and returned after finishing the process. The Figure 3 presents the overview of the verifier model.

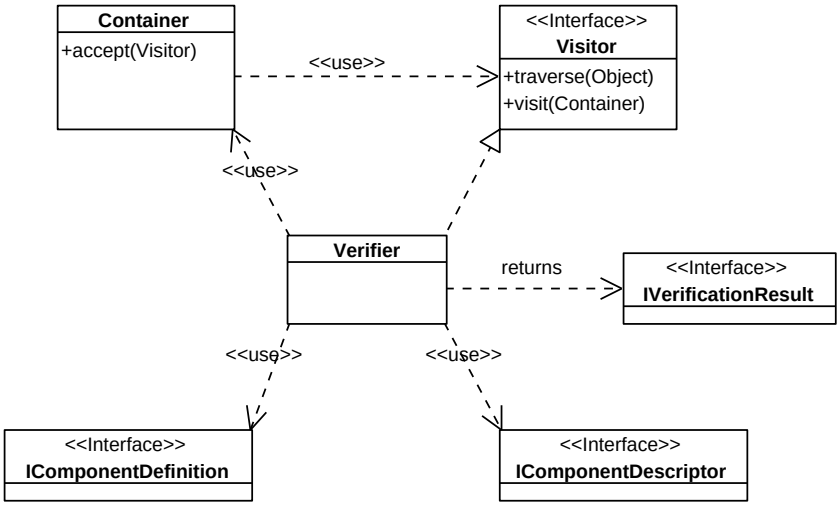

Figure 3. Verifier UML scheme.

In fact, the verification process is performed by independent verification modules which are attached to the verifier.

Each independent verification module gets a single component definition, the IoC container, and a results queue, and then verifies if the component can be properly instantiated according to rules defined by the module.

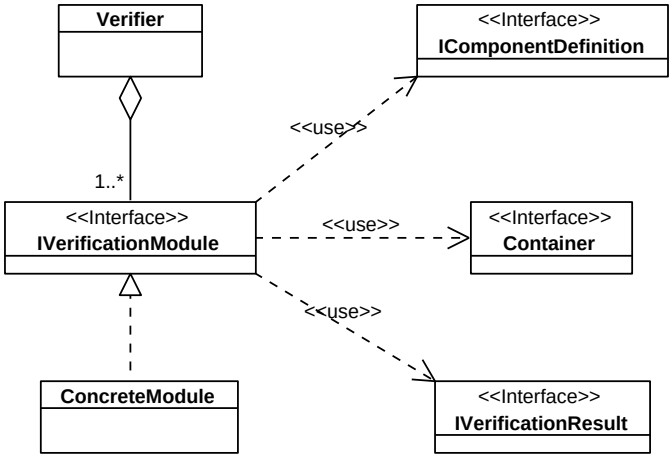

Figure 4. Verifier modules UML scheme.

We propose the following default verification modules that verify the basic rules described below:

- `ComponentLackVerificationModule` – verifies if a component definition provides all the component dependencies described in a component descriptor.
- `DependenciesTypeVerificationModule` – verifies if a component definition provides components whose types are consistent with those described in a component descriptor,
- `DependenciesExistVerificationModule` – determines recursive verification; component is verified negatively if its dependencies are verified negatively, even if they exist and their types are consistent,
- `DependenciesCycleVerificationModule` – checks if there are any cyclic dependencies; the module should contain `DependenciesExistVerificationModule`,
- `ConstructorVerificationModule` – verifies if a constructor used in a component definition is consistent with those described in a component descriptor.

Verification modules can be arbitrarily attached to the verifier and can also constitute a hierarchical structure. The sample modules configuration is shown at the Figure 5.

The presented configuration implies a process started by the verifier. It visits each component definition and invokes the `verify` method from `DependenciesCycleVerificationModule`. Afterwards, the `DependenciesCycle-`

`VerificationModule` invokes verify methods sequentially from the child modules. The `DependenciesExistVerificationModule` module, before finishing its process, passes the Visitor to all dependent components. Module configuration can be constructed differently and it does not change the result. The only requirement is that `DependenciesExistVerificationModule` must be a child module of `DependenciesCycleVerificationModule`.

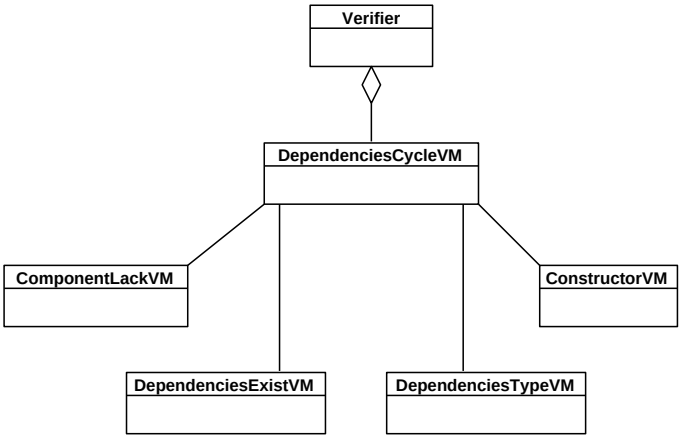

Figure 5. Verifier Built-in modules.

Each verification module adds its result to the `IVerificationResult` object, which is returned from the verification process. Afterwards, there is aggregated information about all errors that occurred during the verification process.

The module architecture described above provides a configurable and extensible method for the verification process.

7. Integration with PicoContainer

PicoContainer provides a verification mechanism based on the *Visitor* design pattern. Each PicoContainer instance has a hierarchy of components, and the visitor walks down this logical hierarchy, starting from the component on the top. Class *VerifyingVisitor* is implementing *Visitor's* pattern, and the method *traverse()* is used for crawling. Each component provides an abstract method *verify()* which is invoked by the visitor on each one. Details can be found at Figure 6.

The proposed Component Verification solution is based on the mechanism prepared by PicoContainer. Main class *Verifier* extends the *VerifyingVisitor* class and realizes the *Visitor* design pattern.

The mechanism of Component Verification is a multi-level process due to its module architecture. The *verify()* method is, in fact, a set of verification functions (from each module, one function is applied). Such a mechanism provides a complex

and complete process of configuration validation. Moreover, the mentioned modularity guarantees flexibility in choosing the needed type of verification. For example, different verification modules may be applied for different component systems.

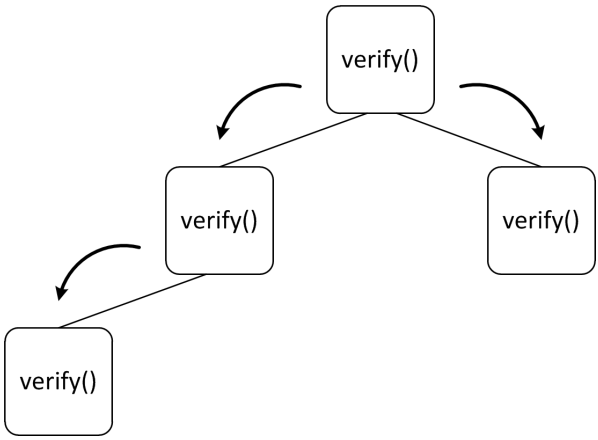

Figure 6. Integration with PicoContainer.

8. Case study

The main purpose of the AgE component platform is to execute distributed agent-based simulations and computations. An exemplary computation is to find a global minimum of a given function using the vanilla genetic algorithm.

In order to run a computation, the user has to provide the platform configuration in the form of an XML file. A part of a sample configuration is presented in the Listing 1.

```
<strategy name="operator" class="org.jage.realvalued.RastriginLocalOperator" />
<strategy name="evaluator" class="org.jage.evaluator.LocalPopulationEvaluator" >
  <constructor-arg>
    <reference target="operator" />
  </constructor-arg>
</strategy>
```

Listing 1: A fragment of the AgE XML configuration file

We can see that the *LocalPopulationEvaluator* component is dependent on the *SolutionOperator* component (for which the *RastriginLocalOperator* implementation of the *SolutionOperator* interface is provided).

If the user made a mistake in the configuration (e.g. a component dependency is missing), the platform will crash, leaving an unclear message containing the stack trace with the *NullPointerException* on top. Assume that the line number 1 is missing

in the Listing 1. Without the verification module, the application crashes, and the result is presented in the Listing 2.

The component verification module checks dependencies before they are used, so the error message is clear (See Listing 3).

```
Exception in thread "main" java.lang.NullPointerException
  at org.jage.agent.GeneticActionDrivenAgent.getResultLog(ActionDrivenAgent.java:124)
  at org.jage.agent.GeneticActionDrivenAgent.finish(ActionDrivenAgent.java:117)
  at org.jage.agent.SimpleAggregate$1.run(SimpleAggregate.java:136)
  ...
```

Listing 2: Missing dependency – AgE crashes (no component verification)

```
ERROR pico.PicoInstanceProvider -- Container verification failed. 4 verification errors:
  Verification error for 'evaluator' of type 'class org.jage.evaluator.LocalPopulationEvaluator':
  unsatisfied dependency 'operator' in PicoInstanceProvider{id=27994965, parent=2622421}
  ...
```

Listing 3: Missing dependency – clear message (produced by component verification)

Component dependencies verification helps a lot while debugging an XML configuration file. It provides a clear message that contains all errors that have occurred (instead of a *NullPointerException* – only for the first failure of the dependency resolution). The user immediately becomes aware of all mistakes which need to be corrected.

The case described above is not the only benefit the user can gain while using the component verification module for the vanilla genetic algorithm. Not less important is the „Fast-fail“ approach. To understand this, it is important to present how the computation is realized inside the AgE platform.

During its activity, the platform iterates over agents and executes the *step()* method. Inside this method, an agent performs an algorithm expressed by the sequence of actions. Each action is a strategy component which represents a part of the algorithm. In the vanilla genetic problem, for example, an agent performs the following actions:

1. Preselection Action,
2. Variation Action,
3. Evaluation Action,
4. Statistics Update Action.

Each action is a strategy component which can have its own dependencies. Assume that the Evaluation Action takes a long time to perform and that the Statistics Update Action has an unsatisfied dependency. Without the verification module, the user would wait a long time just to receive a *NullPointerException*. The verification module fulfills the „Fast-fail” paradigm and provides an error message before the long computation starts.

9. Conclusions

The main advantage of the proposed Component Verification System is early error detection. The first stage of verification is performed before the components are instantiated. This prevents the whole system from failures in execution time. What is more, the verification process does not stop after the first error is found – all of the rules and possible errors are checked. In the result, a list of all faults is returned. This allows us to fix the majority of errors at once instead of correcting them one at a time.

Verification System is also easily extensible. In order to extend its functionality, the only required action is to create and plug in the new *VerificationModule* (for details see section 6). Component Verification is designed for systems based on the *IoC pattern*, which is a common idea for Spring and PicoContainer frameworks. Due to this, Component Verification might be applied (with minor changes) to the Spring- and PicoContainer-based systems.

The main disadvantage of the proposed component verification solution is the fact that it is not versatile. Despite the possibility of generalization to other component platforms (Spring-, PicoContainer-based), the current version works only for the AgE platform. In future work, we hope to achieve versatility: the verification module should be applicable to any component environment.

References

[1] Agrawal A.: Service-Oriented Architecture. `http://www.rightwaysolution.com/soa.html`, 2009. Rightway Solution (India) Pvt. Ltd.

[2] Bernard E. et al.: JSR 303: Bean Validation. `http://jcp.org/en/jsr/detail?id=303`, 2009.

[3] Crnkovic I., Henrik M. P.: Building Reliable Component-Based Software Systems. Artech House, 2002.

[4] Erl T.: Service-Oriented Architecture: Concepts, Technology, and Design. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2005. ISBN 0131858580.

[5] Gamma E., Helm R., Johnson R., Vlissides J.: *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1998.

[6] Garlan D., Shaw M.: An introduction to Software Architectures. 1994. School of Computer Science, Carnegie Mellon University, Pittsburgh.

[7] Hall R. S., Pauls K., McCulloch S., Savage D.: OSGi in Action. Manning Publications, 2011.

[8] Lee B., Johnson R.: Dependency Injection for Java. `http://jcp.org/aboutJava/communityprocess/final/jsr330/index.html`, 2009.

[9] Liskov B., Wing J.: A Behavioral Notion of Subtyping. vol. 16, 1994.

[10] Martin R.: Clean Code: A Handbook of Agile Software Craftsmanship. Prentice Hall, 2008.

[11] Piętak K., Kisiel-Dorohinicki M.: Agent-Based Framework Facilitating Component-Based Implementation of Distributed Computational Intelligence Systems. In: N. T. Nguyen, J. Kołodziej, T. Burczyński, M. Biba, eds., *Transactions on Computational Collective Intelligence X*, *Lecture Notes in Computer Science*, vol. 7776, pp. 31–44. Springer Berlin Heidelberg, 2013. ISBN 978-3-642-38495-0. http://dx.doi.org/10.1007/978-3-642-38496-7_3.

[12] Piętak K., Woś A., Byrski A., Kisiel-Dorohinicki M.: Functional Integrity of Multi-agent Computational System Supported by Component-Based Implementation. In: *Proceedings of the 4th International Conference on Industrial Applications of Holonic and Multi-Agent Systems: Holonic and Multi-Agent Systems for Manufacturing*, HoloMAS '09, pp. 82–91. Springer-Verlag, Berlin, Heidelberg, 2009. ISBN 978-3-642-03666-8. http://dx.doi.org/10.1007/978-3-642-03668-2_8.

[13] Shore J.: Fail Fast. *IEEE Software*, vol. 21(5), pp. 21–25, 2004. http://dblp.uni-trier.de/db/journals/software/software21.html#Shore04a.

[14] Szyperski C.: Component Software: Beyond Object-Oriented Programming. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd ed., 2002. ISBN 0201745720.

Affiliations

Rafał Leszko
AGH University of Science and Technology, Institute of Computer Science, Krakow, Poland,
leszko@agh.edu.pl

Kamil Piętak
AGH University of Science and Technology, Institute of Computer Science, Krakow, Poland,
kpietak@agh.edu.pl

Received: 20.02.2013
Revised: 03.07.2013
Accepted: 03.07.2013