# Enforcing Component Dependency in UML Deployment Diagram for Cloud Applications

Reza Gorgan Mohammadi
Department of Computer Engineering and IT
Amirkabir University of Technology
Tehran, Iran
reza_gorgan@aut.ac.ir

Ahmad Abdollahzadeh Barforoush
Department of Computer Engineering and IT
Amirkabir University of Technology
Tehran, Iran
ahmad@aut.ac.ir

*Abstract*—**Deployment of cloud applications relies upon the inter-dependencies between software components. Modeling languages like UML can be employed to handle the complexities of deploying cloud applications. UML devotes a special model called deployment diagram to represent the physical view of deployment. A recurring issue in deployment is not considering the dependency between artifacts based on the dependencies indicated in the corresponding UML component diagram. In this paper we present a method for enforcing component dependencies in a UML deployment diagram. Models and metamodels are represented using directed graphs and dependency enforcement is performed using graph transformation. The proposed method has been implemented upon VIATRA2 model transformation framework which illustrates the feasibility of the method.**

***Keywords***— ***cloud application; UML; component diagram; deployment diagram; dependency enforcement***

## I. Introduction

Cloud computing, as a new paradigm, offers a relatively cheap and flexible solution for the deployment of software applications. Software deployment constitutes all activities to make a software system available to its users in the corresponding environment. Software deployment is often a large and complex process due to heterogeneous technologies and various components embodied in the system. The inter-dependencies among software components significantly increase the complexity of the deployment process [1].

The use of models can help tackling the complexities of application deployment by raising the level of abstraction. UML is a de-facto standard for modeling software systems which provides a set of models each capturing a special view of the system under study [2]. The components of the system and the inter-dependencies among components are represented using a component diagram. The physical aspects of a software system can be illustrated using UML deployment diagram.

An important issue for successful deployment of applications in cloud is considering the dependencies among components. The absence of required components may result in the failure of the deployment. Such dependencies are indicated by UML component diagram which should be available in the corresponding artifacts in the deployment diagram. Figure 1 illustrates a motivating example where the dependencies in component diagram are not present in the related deployment diagram. In order to better illustrate the case we have used a diagram called *manifestation diagram* which shows manifestation of components by artifacts. As can be seen in the figure, there are three violation cases in the deployment diagram. In this work we assume that artifacts are deployed on the same virtual node.

In this paper we propose a method for enforcing dependencies defined in the UML component diagram in the corresponding deployment diagram. Using the method we can prevent deployment failures in cloud. Models can be represented using directed graphs. Each model should conform to a metamodel which is itself a special directed graph called type graph. Any violations of dependency conditions are detected as a pattern in the graph. The enforcement is realized using graph transformations which transform the original graph into another graph satisfying the dependencies.

The method proposed in this paper has been implemented upon VIATRA2 model transformation framework [3]. It provides facilities for graph pattern matching and graph transformation. The framework employs a uniform model representation in a model space which makes it applicable to UML models using importers and exporters.

The remainder of the paper is structured as follows. First, in Section II we give the preliminary definitions. Section III presents the proposed method for dependency enforcement in UML deployment diagram. Section IV concerns the implementation issues. The related works are reviewed in Section V. And finally, Section VI concludes the paper.

## II. Definitions

In this section we give the major preliminaries and definitions. In this paper we employ graph representation for models. A directed graph is a set of nodes (vertices) connected by directed edges.

**Definition 1 (Directed Graph).** A directed graph $G = (V, E, src, trg)$ consists of a set V of vertices and a set E of

978-1-4799-5359-2/14/$31.00 ©2014 IEEE

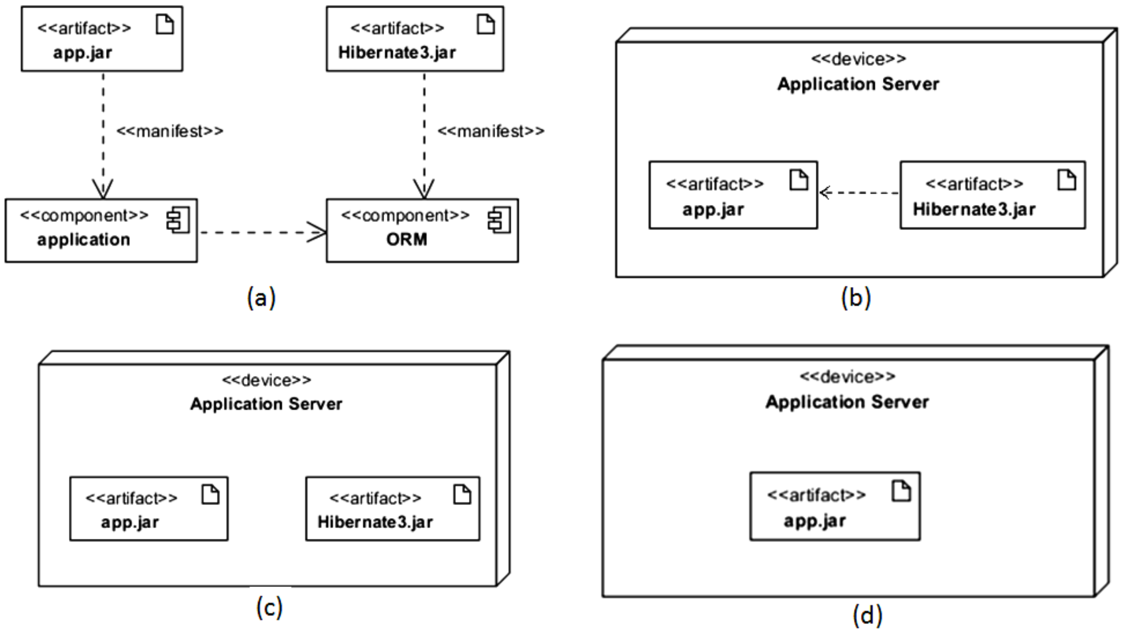

Figure 1. Motivating example. (a) The manifestation diagram (b) a deployment diagram with wrong dependency direction between artifacts (c) a deployment diagram with no dependency relation (d) the deployment diagram with a missing artifact.

edges. The two functions $\mathrm{src}, \mathrm{trg}: \mathrm{E} \rightarrow \mathrm{V}$ determine the source and target of each edge respectively.

**Definition 2 (Graph Morphism).** Given two graphs $\mathrm{G} = (\mathrm{V_G}, \mathrm{E_G}, \mathrm{src_G}, \mathrm{trg_G})$ and $\mathrm{H} = (\mathrm{V_H}, \mathrm{E_H}, \mathrm{src_H}, \mathrm{trg_H})$, a graph morphism $\mathrm{m}: \mathrm{G} \rightarrow \mathrm{H}$ consists of a pair of functions $\mathrm{m_V}: \mathrm{V_G} \rightarrow \mathrm{V_H}$ and $\mathrm{m_E}: \mathrm{E_G} \rightarrow \mathrm{E_H}$ that preserve the source and target of each edge, i.e., $\mathrm{src_H} \circ \mathrm{m_E} = \mathrm{m_V} \circ \mathrm{src_G}$ and $\mathrm{trg_H} \circ \mathrm{m_E} = \mathrm{m_V} \circ \mathrm{trg_G}$.

The vertices and edges of a graph can be assigned types which form a type graph [4].

**Definition 3 (Type Graph and Typed Graph).** A type graph (TG) is a directed graph where $\mathrm{V_{TG}}, \mathrm{E_{TG}}$ are called the vertex and the edge type alphabet respectively. A typed graph is a tuple $(\mathrm{G}, \mathrm{type})$ where G is an instance graph typed over TG by a typing morphism $\mathrm{type}: \mathrm{G} \rightarrow \mathrm{TG}$.

Using the definitions above, we can define a model and its corresponding metamodel. A model is considered as a directed graph typed over the metamodel (a type graph). A graph morphism (mapping) is used to relate a model with its metamodel.

**Definition 4 (Metamodel and Model).** A metamodel is a type graph $\mathrm{G_M}$ and a model m is an instance of the metamodel which is a pair $\mathrm{m} = (\mathrm{D_m}, \mathrm{t_m})$ where $\mathrm{D_m}$ is a directed graph typed over $\mathrm{G_M}$ using the typing morphism $\mathrm{t_m}: \mathrm{D_m} \rightarrow \mathrm{G_M}$.

## III. Dependency Enforcement

UML deployment diagram is used to represent the deployment view of an application. Figure 2 illustrates a portion of metamodel of deployment diagram in UML 2. It also includes component and manifestation items which are from the metamodel of the component diagram.

According to the definitions in the previous section, a model and its corresponding metamodel can be represented using directed graphs. The graph representation of a portion of metamodel in figure 2 is given in figure 3. Part (a) of the figure is related to the deployment metamodel and part (b) illustrates the graph representation of the metamodel of component diagram. The *"artifact"* element is common in both graphs and is an overlap between two diagrams.

The graph representation of manifestation diagram in the motivating example (figure 1-a) is shown in figure 4. The deployment diagrams in the figure (parts b, c, d) can be similarly represented using directed graphs. The dependency relation illustrated in the graph should be available in the deployment diagram. In this paper we employ the artifact entity as the overlap between UML component diagram and deployment diagram. In other words, for each artifact in the manifestation diagram, there should be a match in the corresponding deployment diagram and the dependencies between components (if any) should be preserved in the deployment diagram.

In order to enforce the dependencies in the deployment diagram we use graph transformation which converts a given graph G to a graph H denoted by $G \Rightarrow H$. A graph production rule $p: (L \overset{l}{\leftarrow} K \overset{r}{\rightarrow} R)$ consists of injective typed graph morphisms $l: K \rightarrow L$ and $r: K \rightarrow R$ where L, K, and R are typed graphs. L and R are called precondition and postcondition of the rule respectively [5].

**Definition 5 (Graph Transformation).** A graph

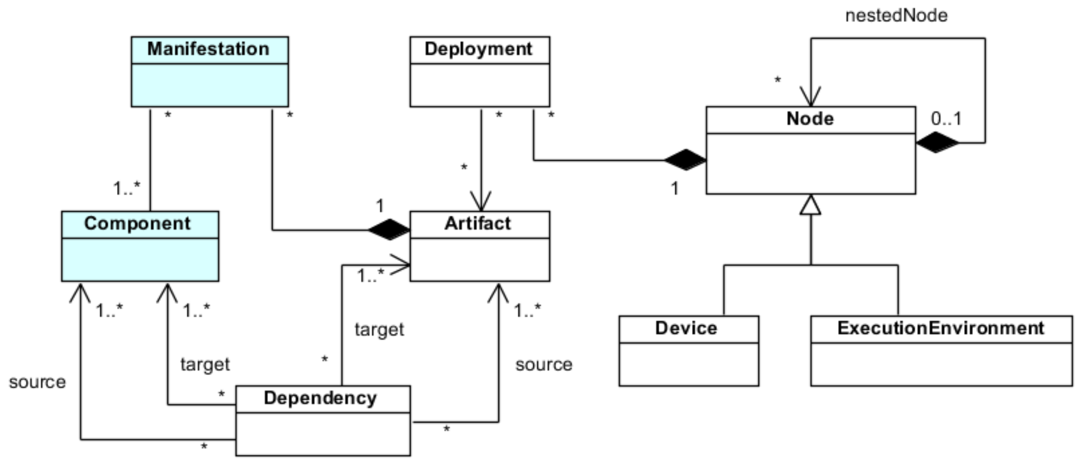

Figure 2. Metamodel of UML 2 deployment diagram with related elements of component diagram.

transformation $G \Rightarrow H$ based on the production rule $p: (L \leftarrow K \rightarrow R)$ and match $m: L \rightarrow G$ converts graph G to graph H. By applying the rule, all the vertices and edges matched by $L \backslash K$ are removed from graph G yielding graph D. The elements of $R \backslash K$ are then appended to graph D which yields graph H.

Application of graph transformation can be restricted using negative application conditions (NACs) which specifies a forbidden graph structure in the host graph.

There are three violation cases which bring about the inconsistency between a component model and its corresponding deployment model. The violations can be considered as graph patterns which form the precondition of graph transformation rules. The deployment diagram may contain the artifacts but not the related dependencies (part a in figure 5). Another condition is where the artifacts and dependencies between them are available in the deployment diagram, however the direction of the dependencies are not correct (part b in figure 5). The last pattern is where an artifact is given in the deployment diagram but the required artifact indicated by the dependency relation is not available (part c in figure 5).

The incorrect deployment model is considered as the host graph which is transformed into another graph satisfying the dependency relations. For each violation pattern the corresponding rule is applied for transformation. For the case of "no dependency relation" (pattern a in figure 5) the rule adds the required dependency. The wrong dependency direction (figure 5-b) is reversed by applying the transformation rule. Finally, the missing artifact and the related dependency are appended to resolve the violation in figure 5-c. The postconditions of all three rules are common and is illustrated in figure 6. Artifacts from deployment diagram and component diagram are shown using "D.artifact" and "C.artifact", respectively.

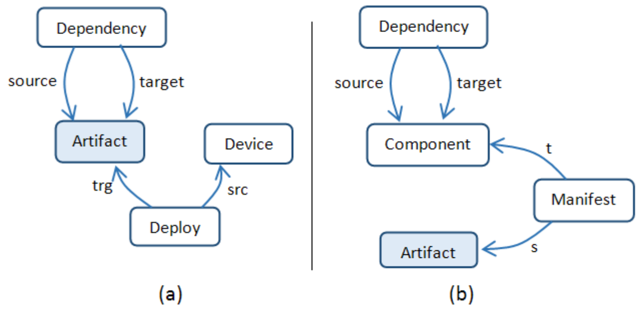

Figure 3. Graph representation of metamodels (key elements). (a) deployment diagram (b) component diagram.

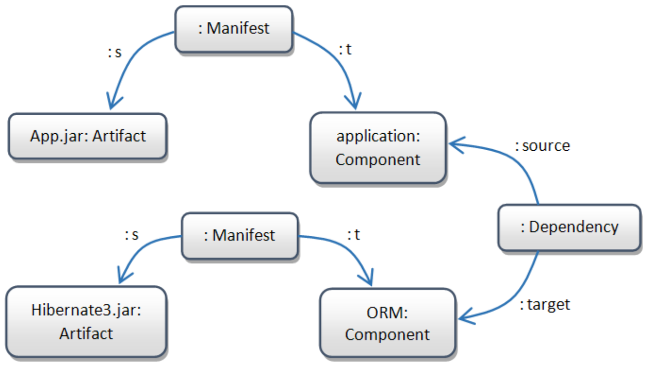

Figure 4. Graph representation of component model

## IV. IMPLEMENTATION

In our research we have employed VIATRA2 model transformation framework for implementation of the proposed method. The framework provides facilities for model transformation based on a uniform model and metamodel representation in a model space. The model space can be populated using importers supporting UML 2 [6].

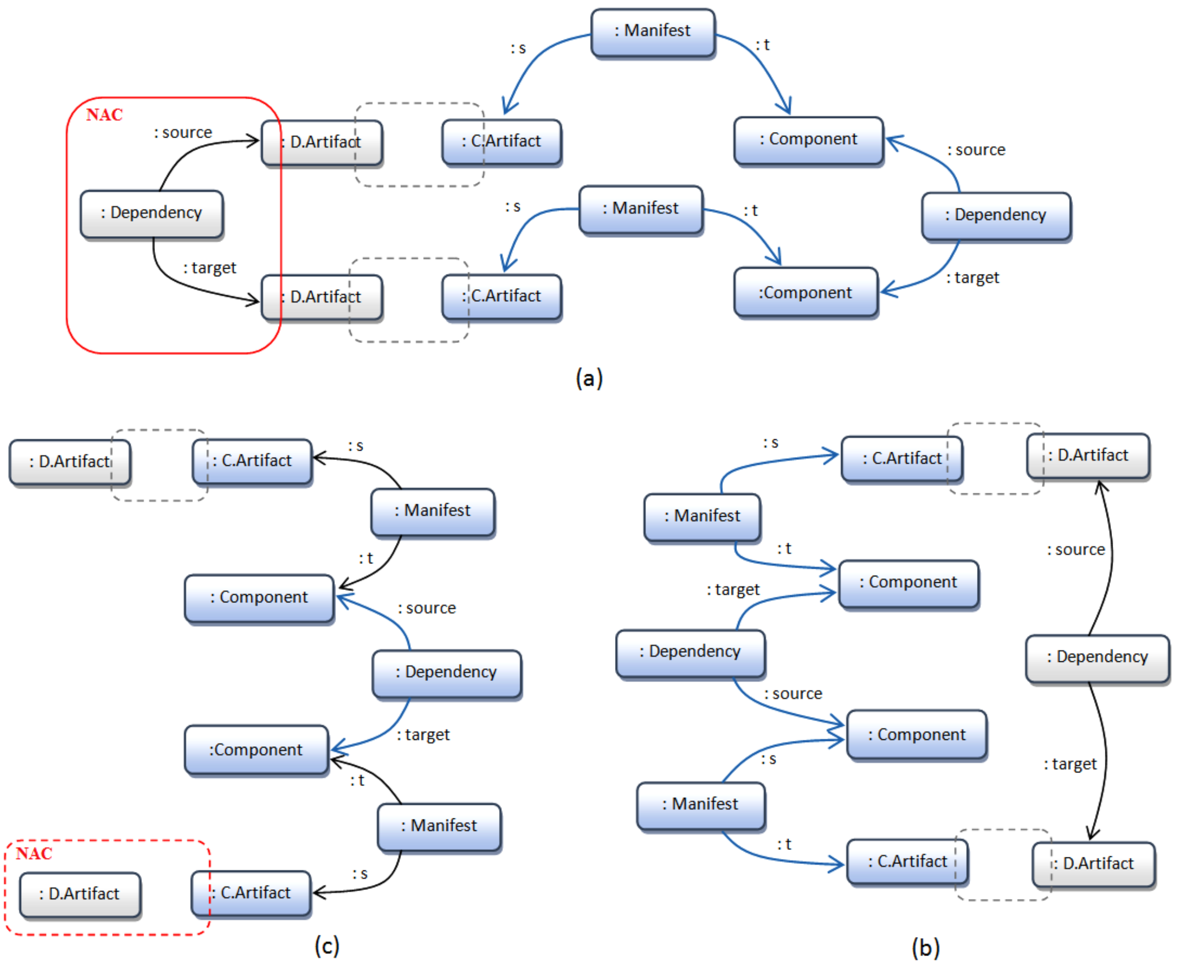

Figure 5. The violation patterns. (a) no dependency relation (b) wrong dependency direction (c) missing required artifact.

The transformation is specified using VTCL language which provides both declarative and imperative features based upon graph transformation (GT) and abstract state machines (ASMs) [3].

The transformation rules for enforcing dependencies in UML deployment diagram have been specified using VTCL language in VIATRA2 framework. As an example the rule for detecting "no dependency" condition is given in Appendix A. The rule in listing 2 specifies the precondition (figure 5-a) and postcondition (figure 6) in order to enforce the dependency. The required patterns for the rule are given in listing 1. We assume that artifacts are deployed on the same virtual node. Also we assign the same value to correlated artifacts in order to find a match between them.

## V. Related Works

In [7] a review of different cloud deployment models is provided. A self-deployment protocol for supporting virtual machine and network failures in cloud applications is addressed in [8].

Zhang et al. [9] propose a new method for user-level virtualization which improves the deployment flexibility. The method isolates application from the OS and VM which allows flexible application deployment in cloud.

Another related branch of research is consistency management of models and multi-models. In [4] a framework is proposed for merging multiple views using a merge operator. Kolovos et.al [10] applies pattern matching in order to merge heterogeneous models. Diskin et al. [11] present a method for finding overlaps between heterogeneous models based on the concepts of category theory. The proposed method in this paper uses the same merge technique based on artifact element to relate a component and a deployment model. In [12] a method for quick fix generation for models is proposed. The method is based on graph representation and

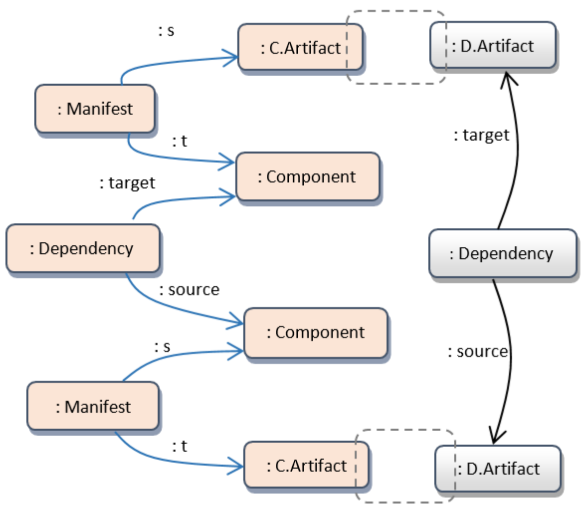

Figure 6. The postcondition of the graph transformation rules.

detects violations as graph patterns. Also the resolution process is performed using graph transformation upon VIATRA2 framework.

## VI. Conclusions

Modeling the deployment aspects of software applications can reduce the complexities. The dependency between components can be modeled using UML component diagram and should be considered in the corresponding artifacts in the deployment diagram. This paper provides a method for enforcing artifact dependencies in UML deployment diagram. The method prevents failures in deployment of applications in cloud.

The proposed method employs directed graphs to represent models and metamodels. The violations of dependency relations are detected as graph patterns and for each kind of violation the corresponding graph transformation rule is applied for dependency enforcement.

## References

[1] S. Patig, "Modeling deployment of enterprise applications," Information Systems Evolution, Springer Berlin Heidelberg, pp. 253-266, 2011.

[2] UML. OMG Unified Modeling Language spec. http://www.omg.org/spec/UML/, 2011.

[3] D. Varró, A. Balogh, "The model transformation language of the VIATRA2 Framework," Sci. Comput. Program. vol. 68, no. 3, pp. 214–234, 2007.

[4] M. Sabetzadeh, S. Nejati, S. Liaskos, S. M. Easterbrook, M. Chechik, "Consistency checking of conceptual models via model merging," RE conference, IEEE, 2007, pp. 221-230.

[5] H. Ehrig, "Fundamental theory for typed attributed graphs and graph transformation based on adhesive HLR categories," *Fundamenta Informaticae, vol.* 74, no. 1, 2006, pp. 31-61.

[6] D. Varró, A. Pataricza, "VPM: A visual, precise and multilevel metamodeling framework for describing mathematical domains and UML," Journal of Software and System Model, vol. 2, no. 3, pp. 187–210, 2003.

[7] S. V. Nandgaonkar, and A. B. Raut, "A comprehensive study on cloud computing," vol. 3, issue. 4, pp. 733 – 738, 2014.

[8] X. Etchevers, et al., "Reliable self-deployment of cloud applications," SAC 2014-29th ACM Symposium on Applied Computing, 2014.

[9] Y. Zhang, Y. Li, and W. Zheng, "Automatic software deployment using user-level virtualization for cloud-computing," *Future Generation Computer Systems*, vol. *29*, no. 1, pp. 323-329, 2013.

[10] D. Kolovos, R. Paige, F. Polack, "Detecting and repairing inconsistencies across heterogeneous models," ICST 2008, Proceedingsof the 2008 International Conference on Software Testing, Verification, and Validation, Washington, DC, USA, 2008, pp. 356–364.

[11] Z. Diskin, Y. Xiong, K. Czarnecki, "Specifying overlaps of heterogeneous models for global consistency checking," Models in Software Engineering, Workshops and Symposia at MODELS, 2010, pp. 42-51.

[12] A. Hegedus, et al., "Quick fix generation for DSMLs," IEEE Symposium on Visual Languages and Human-Centric Computing, VL/HCC, Pittsbourgh, PA, USA, 2011, pp. 17-24.

## Appendix A. VTCL Source Code

The source code of implementation using VTCL language is given in the following listings.

```
Pattern ManifestDiagram(A,B,C,D,E,F,G) = {
  ComponentDiagram.Component(A);
  ComponentDiagram.Manifest(B);
  ComponentDiagram.Artifact(C);
  ComponentDiagram.Manifest.s(X,B,C);
  ComponentDiagram.Manifest.t(Y,B,A);
  ComponentDiagram.Component(D);
  ComponentDiagram.Manifest(E);
  ComponentDiagram.Artifact(F);
  ComponentDiagram.Manifest.s(K,E,F);
  ComponentDiagram.Manifest.t(L,E,D);
  ComponentDiagram.Dependency(G);
  ComponentDiagram.Dependency.source(P,G,A);
  ComponentDiagram.Dependency.target(Q,G,D);
  }

Pattern ArtifactMatch(A,B,C,D)= {
      ComponentDiagram.Artifact(A);
      DeploymentDiagram.Artifact(B);
      check (value(A) == value(B));
      ComponentDiagram.Artifact(C);
      DeploymentDiagram.Artifact(D);
      check (value(C) == value(D));
  }


Pattern NoDep (D,H,Z)={
    DeploymentDiagram.Artifact(D);
    DeploymentDiagram.Artifact(H);
    DeploymentDiagram.Dependency(Z);
    DeploymentDiagram.Dependency.source(P,Z,D);
    DeploymentDiagram.Dependency.target(Q,Z,H);
 }
```

Listing 1. Specification of common patterns

```
   Gtrule CheckNoDependency (out A, out B, out C,
out D, out E, out F, out G, out H, out I )= {

   Precondition pattern NoDependency
(A,B,C,D,E,F,G,H,I)= {
       find ManifestDiagram (A,B,C,D,E,F,G);
       find ArtifactMatch (C,H,F,I);
       neg find NoDep (D,H,Z);
   }

   action {
     println("no dependency between artifacts");
     let R = undef, U=undef , V=undef in seq {
          new (DeploymentDiagram.Dependency(R)in
       SourceModel.Deployment);
          new (DeploymentDiagram.Dependency.
       source(U,R,D));
          new (DeploymentDiagram.Dependency.
       target(V,R,H));
         }
      }
   }
```
Listing 2. Specification of “no dependency” pattern