Building a model for analysis of dependencies of application software modules

Gorbenko O D and Kochergin D V
Department of Applied Mathematics, Informatics and Mechanics, Voronezh State University, 1, University Square, Voronezh, 394018, Russia
E-mail: dkochergin17@gmail.com

Abstract. Dependency analysis of software modules in applications is an important step in software development, but it almost always remains in the shadow of more general and frequent problems. The creation of a general mathematical model describing dependency analysis at a fundamental level is one of the tasks that can accelerate the development of new utilities for dependency analysis for a wide variety of module systems and programming languages.

1. Introduction
Modern software products are complex systems that include a large code base, usually divided into modules that are linked by dependencies. The structure of these modules, as well as the ways to organize dependencies between them, can affect both the architecture and the stability of the software product, its fault tolerance.

Dependency management is an important process that begins with writing the very first modules in a program, and continues until the end of the software development cycle, which can be more complex. Analysis of such dependencies allows you to speed up the process of developing a software product, as well as avoid many problems that usually arise at different stages of development due to insufficiently thought-out organization of dependencies of software modules of the program being developed. The subject of analysis can be both finding cyclic dependencies, and, for example, code refactoring or clustering the dependency graph using various algorithms [1, 2].

In this article, we developed a dependency graph model that allows us to analyze the structure of modules of various applications. In addition, one of the tasks of the model is a general description of module dependencies without references to specific programming languages.

The model has a number of features:

- Creating a dependency graph;
- Finding cyclic dependencies between modules;
- Finding unused or isolated modules in the dependency graph.

Based on this model, you can also create software that allows you to analyze software modules for different programming languages. An example of such software will be discussed later.
2. Materials and methods

Currently, no well-formed algorithms and models have been found that allow analyzing the dependency graph. Such algorithms are used in build systems and task runners for various programming languages, but they are not formalized and are designed to meet the requirements and capabilities of a specific programming language, which does not allow us to consider them at a fundamental level in isolation from a specific programming language. To do this, we need to consider the basic concepts and definitions used in the model.

2.1. Basic concepts and definitions used in the model

The main concepts around which the model is built are the module and the dependency graph. The dependency graph is built on the basis of modules, and then the model works only with the dependency graph.

A module is an entity that abstracts a part of the application and refers to other similar entities that can also refer to this one. The entities that the module refers to are called imports, and the set of entities that refer to this module is called exports. In addition to links to other entities, a module can contain additional information about the links themselves and the modules they point to.

A dependency graph of an application (software product) is a directed multigraph without loops, with possible hanging and isolated vertices, such that:

- Each vertex of the graph is assigned an unambiguous correspondence to a certain software module from the application under consideration;
- Each directed arc in the graph represents a set of dependencies of a particular module and is uniquely associated with this set.

![Figure 1. Simple visualization of a dependency graph.](image)

2.2. Algorithm for constructing the dependency graph

The dependency graph is formed on the basis of a set of modules in two stages:

(1) The dependency graph is formed on the basis of a set of modules.
(2) Saturate the graph with additional information about the use of modules and their specific dependencies.
At the first stage, the main structure of the graph is constructed using links between some modules and others. Connections between modules become arcs in the oriented graph, and the modules themselves become vertices (as it follows from the definition of the dependency graph).

There are several ways to construct arcs in the graph at this stage, differing in whether you can connect two vertices A and B with only two arcs (from A to B and Vice versa), or whether the number of arcs between two vertices is unlimited.

In the case of the first method, in the dependency graph, each arc from a to b must correspond to a set of dependencies that are imported into module b from module a. This entails the complexity of algorithms for both constructing and analyzing dependencies in general.

To apply the second method, no additional manipulations with the dependency graph are required, each dependency will be associated with a separate arc of the graph. In this case, standard graph algorithms can be used for analysis. For example, search for all simple cycles or isolated vertices.

At the second stage, additional data that could be stored in modules is added to the graph. Most often, this data depends on the specific programming language in which the program is written, as well as on the selected system of modules and dependencies. such data may include information about the use of dependencies within modules, links to the file system (if the module system depends on it), and so on.

After building the dependency graph, further work occurs only with it, since all data about modules and their connections (including all dependencies) are already included in the dependency graph.

The next stage of the model will be the analysis of the dependency graph for the presence or absence of special states of nodes and arcs that describe certain situations in the application under study.

2.3. Analysis of the dependency graph

At the moment, the analysis of the dependency graph is carried out according to several criteria:

(1) Cyclic dependencies in the graph.
(2) Dependence of modules of lower levels on modules of higher levels.
(3) Isolated and unused modules.

Further, these criteria can be expanded as the model improves and new situations in applications are described, tracking which makes sense to improve the speed and convenience of development.

The problem of finding cyclic dependencies in a graph is reduced to finding simple cycles in a directed graph (which is the dependency graph). For this task, many algorithms have been developed that are optimal both in terms of speed and operating time. As an example of one of the fastest, we can cite the algorithm of Donald Johnson [3, 4].

The dependence of lower-level modules on higher-level modules is understood as a situation when functionality is imported into a lower-level module from a module at least one level higher in the file system directory hierarchy. To find such modules, you need to go through all the vertices of the graph, and for each vertex, find the edges that enter this vertex from the vertex associated with the module located in a directory of a higher level than this one. Vertices that have more than zero such edges will be associated with the required modules.

A module is called isolated if it is associated with an isolated vertex in the dependency graph. Finding isolated modules is reduced to finding an isolated vertex in the dependency graph.

A module is called unused if it is not dependent on at least one other module.

Each dependency graph has at least one such module — the entry point module. However, having more than one such module in the system is not a good practice.
To find all modules that fit these criteria, you can go to a weighted dependency graph, where the weight of a vertex is the number of incoming and outgoing arcs. You can also mark vertices with numerical characteristics that correspond separately to the number of incoming and outgoing arcs.

After these operations, the criteria check will be reduced to a numerical check of the vertex weight, as well as its other parameters marked in the graph.

3. Results and discussion

As an example of the implementation of this model, a utility was developed that allows the analysis of ES modules (corresponding to the ECMAScript standard [5]) in JavaScript applications [6].

The utility is installed depending on the target application and analyzes it for some flaws, such as:

- Presence of cyclic dependencies;
- Availability of isolated modules;
- Presence of unused modules.

To generate a dependency graph and analyze it, the utility uses ESPree and ESCope libraries. These libraries allow you to perform lexical and syntactic analysis [7], as well as analyze visibility zones to determine the use of dependencies within the module.

The utility also provides an interactive visualization of the dependency graph with the ability to view information about each module and output a general report on the number of modules and imports, as well as their use.

3.1. Utility structure

The utility structure is based on modules, which allows you to decompose the functionality and divide the areas of responsibility in accordance with the mathematical model.

![Diagram](image)

Figure 2. The utility modules and links between them.
Creating a graph and its content information (i.e., analysis) are posted in different modules, it also allows to allocate a separate entity the process of obtaining a set of modules for a graph and thus make possible the construction of a dependency graph for applications written in different programming languages using various parsers. A diagram describing the main modules of the utility and the connections between them is shown in figure 2.

The main difference in structure from the mathematical model is the presence of project tree generation and visualization modules. The first is a feature of the implementation of the model for a specific programming language – JavaScript. The second one is an addition to the model, allowing you to visualize the results of your work. A detailed review of the visualization of the dependency graph and the results of its analysis is beyond the scope of this article.

3.2. Utility interface
To use the utility, you must install it in the developer’s dependency for the target Node.js application. To do this, use the following command:

```bash
npm install -D dependency-util
```

After successful installation, the utility application can be run from the scripts section of the package.json of the target application. To do this, add the following command to the scripts section:

```bash
du <params>
```

`Params` refers to the utility parameters described in table 1.

| Name    | Type     | Mandatory | Description                                                                 |
|---------|----------|-----------|-----------------------------------------------------------------------------|
| Entry   | Unnamed  | Yes       | The entry point to the app. Represents the path to the root folder of the application. |
| Port    | Named    | No        | The port on which the web application with interactive visualization will be launched. The default value is 3000. |
| BaseUrl | Named    | No        | The base path from which non-relative imports are resolved (the same parameter is available for collection systems). The default value is equal to the value of the Entry parameter. |

After successful launch, the console displays a message:

```bash
du server is listening on 3000
```

This means that you can go to the browser (Chrome is supported) and open the page `http://localhost:3000` to view the dependency graph. After loading, the start page is displayed on the screen with the ability to view individual modules and cyclic dependencies in the graph.
The utility has an interactive graph visualization with general information in the tabs on the left side of the screen. In the *modules* tab, the user can see a list of all modules with the ability to click on the module and view the detailed information. On the *cycles* tab, the user has similar functionality related to cycles, as well as the ability to see cycles on the graph (figure 3). The *report* tab contains all information about the dependency graph.

![Figure 3. Viewing cyclic dependencies in the graph.](image)

4. Conclusion
As a result of this work, the model for performing dependency analysis of software modules was developed and formalized, and the utility application was developed as an example of the implementation of this model. Using this model will allow developers to improve the quality of the applications they create and reduce the time spent on finding flaws in the dependencies of the created applications.

References
[1] Bittencourt R A and Guerrero D D 2009 Comparison of Graph Clustering Algorithms for Recovering Software Architecture Module Views 13th European Conference on Software Maintenance and Reengineering
[2] Lee S, Cho B and Song Y 2010 A Study on Crosscutting Refactoring Using Program Dependency Relation 9th International Conference on Computer and Information Science
[3] Johnson D 1975 Finding All the Elementary Circuits of a Directed Graph SIAM Journal on Computing 4 77-84
[4] Meyer F Search all elementary cycles in a given directed graph URL [http://normalisiert.de/code/java/elementaryCycles.zip](http://normalisiert.de/code/java/elementaryCycles.zip)
[5] TC39, E.C.M.A. 2016 ECMA-262: ECMAScript® 2016 Language Specification Ecma International, Geneva, Switzerland
[6] Flanagan D 2020 JavaScript: The Definitive Guide 7th edn (Sebastopol, CA: O’Reilly)
[7] Aho A, Lam M, Sethi R, Ullman D 2006 Compilers: Principles, Techniques, and Tools 2nd edn (Boston, Massachusetts, USA: Addison-Wesley)