Code Duplication in ROS Launchfiles

Pablo Estefó, Romain Robbes, Johan Fabry
PLEIAD and RyCh labs, Computer Science Department (DCC), University of Chile, Chile
{pestefo,rrobbes,jfabry}@dcc.uchile.cl

***Abstract*— The middleware for robotics ROS has become the de-facto standard for developing robot applications. Thanks to our experience using ROS we conjectured that the quality of software of ROS is low, yielding a poor user experience for ROS users and posing important barriers to robot software development. In this work we present a first quantification of code quality of the ROS ecosystem through an analysis of code duplication in launchfiles. Our experience led us to believe that these configuration files exhibit a large amount of code duplication, and this study shows that it is indeed the case. We find that 25% of packages with multiple launchfiles have duplicated code, and that clones are highly similar.**

I. Introduction

The Robot Operating System (ROS) is a middleware for robots [14]. It provides several libraries and tools for developing complex robotic applications. It is open source licensed and comprehends various projects from different universities and companies all over the world. Developers and Researchers can write *packages* that implement robot behaviour in a wide variety of programming languages. ROS follows the publish-subscribe model for flexible communications and network scalability. In this model, *nodes* are processes that perform computation on certain machines. A node communicates data with other nodes by passing *messages*, a typed data structure. To achieve this, a node publishes them to a *topic* that other nodes may be subscribed to.

ROS is however more than just a middleware: as it is arguably the de-facto standard robotics middleware, a significantly large development effort in robotics research happens on top of ROS. As a result, there are many packages that offer parts of robot behavior built on top of ROS, and we can consider it as a software ecosystem. A Software Ecosystem is often defined as "a collection of software projects which are developed and which co-evolve together in the same environment" [9]. Hence ROS represents a software ecosystem for developing complex robotic applications.

In our experience using ROS to develop behaviors on our robots, we suspected that the software quality of ROS and the various packages in the ROS ecosystem is low. As a result, ROS is hard to install and use, and a significant amount of time is lost using it, which could better be used to perform research. We therefore set out to quantify the code quality of the ROS ecosystem so as to later be able to suggest improvements. We report here on a first investigation on code duplication, as code duplication is one criterion that can be used for establishing code quality. To scope down the investigation, we only consider one type of configuration files in the ecosystem: launch files, as in our experience we have seen many cases of code duplication in these files.

II. The ROS Ecosystem

The ROS Ecosystem consists of its core stack (*e.g.* `roscore`: the process that coordinates nodes and topics, and `catkin` the build system), dedicated tools (*e.g.* `RViz` for 3D visualization, and `rqt_graph` for active nodes and topics visualization) and third party ROS packages.

These ROS packages represent the bulk of this ecosystem, each one representing a specific feature that can be reused by other packages. ROS packages can be developed in multiple languages. The tutorials of ROS show C++ or Python as the main alternatives but ROS also provides bindings to others: Java (ROSJava) [6], Lisp (ROSLisp) [11], and Smalltalk (PhaROS) [4].

On the 25th of March there were 1672 ROS packages registered on the official ROS website (http://www.ros.org/browse/list.php). 87% of the packages are hosted on Github, 8 packages on Bitbucket and 176 of them do not report any hosting site. Our sample considers all of the 469 github repositories, containing 1560 ROS packages with 47796 files totalling 1.73 GB.

	Category	**Number of files**	**%**
Source Code	C++	6238	13.1
	C	5722	12.0
	Python	3524	7.4
	Lisp	1250	2.6
	JavaScript	375	0.8
	Bash	298	0.6
	Java	257	0.5
	Others (eg. ruby, qt)	889	1.8
	Total	**18553**	**38.8**
ROS Related	Launchfiles	2810	5.9
	Others	2757	5.7
	Package definition	1671	3.5
	Message definition	1237	2.6
	Xacro	668	1.4
	Service definition	574	1.2
	Robot Structure	530	1.1
	Total	**10258**	**21.46**
Documentation		**4538**	**9.5**
Build files		**3677**	**7.7**
3D Modeling		**2926**	**6.1**
Pictures		**2333**	**4.9**
Project metadata		**1073**	**2.2**
Non-categorized		**4438**	**9.28**

TABLE I: Categorization of files per use

Special to the ROS ecosystem is that the types of files in each ROS package are quite diverse. They can be the source code of the nodes, build files, message and service type definitions, documentation files, robot configuration files, xml files for launching several nodes (called *launchfiles*) and the mandatory `package.xml` definition file. A semi-automatic categorization of all files was done by mapping the extension to its kind and for files without extension (2.8%) by mapping its name.

The result of this work is shown in Table I. In it we see that the predominant programming languages in ROS packages are C/C++ and Python, covering 32.5% of the files. The second big category are ROS related files (27.6%) which considers *launchfiles* (5.9%), package xml file definitions (3.5%) and message definitions (2.6%). This group is followed by Documentation (9.5%) and 3D Modeling files (6.1%). The latter are required for simulation and robot object perception.

```
<launch>
  <node pkg="turtlesim" type="turtlesim_node" name="sim" />
  <param name="publish_frequency" type="double" value="10.0"
        />
  <include file="\$(find other−pkg)/path/turtlebot−spec.xml" />
</launch>
```

Listing 1: ROS launchfile example taken from ROS Tutorials and modified.

The amount of non-categorized files could not be reduced because of the high variety of extensions (481), the most frequent extensions in this category had no more than 20 files and 462 extensions had 10 files or less.

The table exposes a high diversity both in file types and technologies involved for implementing robot tasks with ROS. This implies that the ROS middleware works with a wide variety of tools that manage different types of files for different uses. As a result we can say that the ROS Ecosystem presents a high heterogeneity, which represents an additional challenge for its study.

III. ROS LAUNCHFILES

ROS provides a way to launch several nodes at once, locally or on several machines, and to set global parameters. To do this, it reads a launch configuration file (a file with the `.launch` extension) that is in XML syntax. These files are also called *launchfiles*.

For example the code in Listing 1 starts the node `turtlesim_node` on the second line. This node is defined in the `turtlesim` package and it is given the name `sim`. The third line defines the parameter `publish_frequency` with type `double` and a value of `10.0`. This parameter can be accessed by the `turtlesim_node` and it is also available for any other node that is launched afterwards. Finally, an external launchfile is included into the current launchfile. All the nodes, parameters and even other included files declared in the file `turtlebot-spec.xml` from the `other-pkg` package are included as if they were defined in this code.

In the ROS ecosystem, from all 1560 packages, 1027 (65.83%) defined no launchfiles. The rest (533) contained 2650 launchfiles (160 are for templating or testing purposes and were ignored), half of these define only one. In terms of distribution, the vast majority (80.39%) defines up to 5 launchfiles, 96 packages (18.01%) define between 6 and 20 launchfiles. There are two outliers (0.43%). The first one, `cob_bringup`, collects all the scripts, launchfiles and dependencies to boot the *Care-O-bot*[1] (59 launchfiles). The second outlier is `jsk_pcl_ros` which provides programs for object recognition, it contains 72 launchfiles.

IV. Clone Analysis of Launchfiles

Copying and then pasting a fragment of source code for reuse is a common practice. [16]. The code fragment could be left as is or might be edited afterwards. In any case, this portion of source code is called a *clone* [7] and considered a bad practice. Its consequences can be enabling bug propagation [8] or design flaws [13], which increases maintenance costs and impacts negatively on evolution. Detecting code clones is a recommended first step for reducing these negative repercussions [5], [16].

A. Number of Clones and Their Similarity

We performed an clone analysis on ROS *launchfiles* as a first investigation of the code quality of the ROS ecosystem. Our strategy was to detect the presence of clones between launchfiles belonging to the same package. For all packages we compared all pairs of launchfiles in the package. We set a threshold of what is considered a clone in order to reduce noise: a pair of launchfiles are considered as a clone pair if they have at least 7 identical lines in common. We found that from all 533 packages, 133 (24.95%) of them have clones. 110 of those (82.7%) contain 6 or less clones, almost a half of packages present one clone (49.62%) and 21.23% of packages present two or three clones among their launchfiles.

Moreover, the packages with less than 6 clones present a high similarity (in average) between the launchfiles involved in clone pairs. The measure of similarity in a clone pair is called *overlap* [2]: let L_a be the set with lines of a launchfile and the operator $|L_a|$ the number of lines in a launchfile, then the overlap operator is defined as follows:

$$\text{overlap}(L_a, L_b) = \frac{|L_a \cap L_b|}{|L_a \cup L_b|} \quad \in [0, 1]$$

[1] http://wiki.ros.org/Robots/Care-O-bot

The more lines two launchfiles have in common, the greater their overlap mesure is and as a consequence, the more similar those launchfiles are. Figure 1 depicts the frequency of packages with a certain similarity (on average) between their launchfiles (that belong to a clone pair). We consider that the average per package is reasonable because for each package there are few clones. A big portion of those packages have similarity of 45% or more. This implies that it is not rare that developers do reuse relevant code fragments in launchfiles by copy-and-pasting.

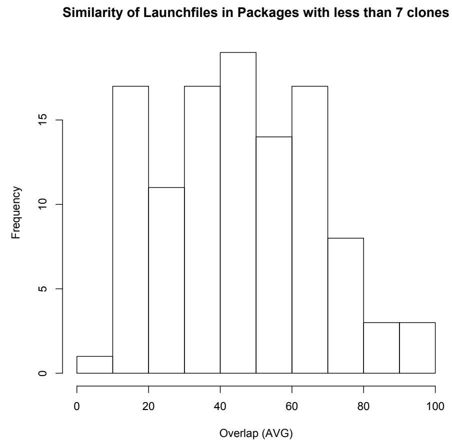

Fig. 1: How similar are the launchfiles in packages with few clones.

B. In-depth Study

In order to better understand how are launchfile code fragments we studied in depth a subset of packages with 7 or more clone pairs. To prioritize which packages to study we used three metrics: the average overlap between clone pairs, the proportion of launchfiles in clone pairs versus all the clones defined in that package (included those which does not belong to any clone pair), and *clone cohesion*. The last metric refers to groups of launchfiles that have many clone relationship between them, and it is calculated as the ratio between the number of clones and number of launchfiles involved in clone pairs. The bigger this ratio is, there is more chance to find shared clone fragments between launchfiles.

TABLE II: Packages with more intersting clone cases

Package	Files	Launchfiles	Launchfiles with Clones	Clones	Average Overlap	Clone Cohesion
(a) hector_quadrotor_gazebo	20	10	8	28	68.86	3.5
(b) jsk_interactive_marker	128	24	11	31	51	2.82
(c) fanuc_lrmate200ic_support	65	21	10	45	50.29	4.5
(d) ueye_cam	22	5	5	10	49.76	2
(e) amcl	43	13	13	78	46.3	6
(f) openni_launch	15	11	7	21	30.66	3
(g) cob_bringup	77	72	19	71	30.4	3.74
(h) cob_controller_configuration_gazebo	18	11	11	55	30.17	5
(i) jsk_teleop_joy	70	12	8	25	25.32	3.13
(j) rtabmap_ros	214	36	31	112	21.59	3.61

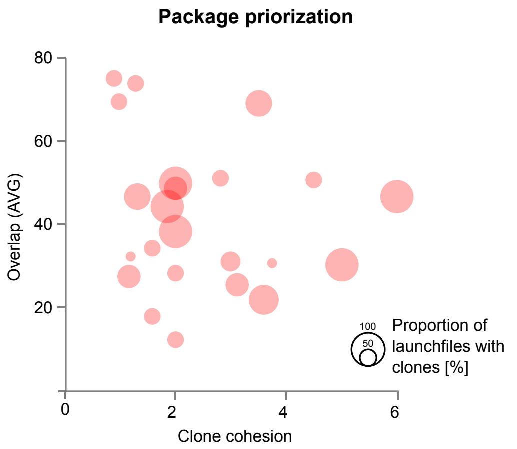

Fig. 2: Packages with more than 7 clones

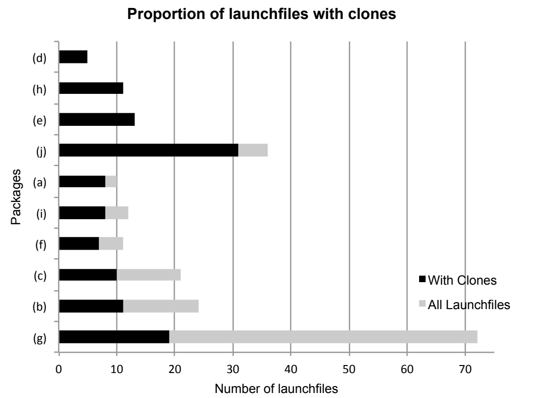

Fig. 3: Proportion of Lauchfiles with clones per package, ordered by proportion of number of launchfiles with clones.

Those three metrics are visualized (see Figure 2) on packages with at least 7 clones, which is an amount that ensures the presence of at least 5 launchfiles in clone pairs. The big circles represent packages with big portion of their launchfiles involved in copy-and-paste activities. Also, the packages located away from the origin and relatively equidistant from both axes are interesting as they represent cases of several launchfiles that share big portions of code and moreover this fragment is repeated in all of them with minor differences.

We made a manual revision and priorization of packages considering all metrics previously described and selected the 10 most relevant cases. These packages are presented in Table II.

Figure 3 shows how many launchfiles in the packages are involved in clone pairs. *(h)* and *(e)* are two packages that have more than 10 launchfiles and all of them present clones. *(j)* contains a vast amount of launchfiles and over 86% of them present clones. *(g)* presents less proportion of launchfiles with clones, but as mentioned in Section III, its amount of launchfiles is far over the average and the amount of launchfiles involved in clones and their cohesion are both high.

Figure 4 illustrates what portion of the launchfiles with clones is the code that is shared between launchfiles, on average. In *(a)*, *(b)*, *(c)* *(d)* and *(e)* nearly 50% of the size of the launchfiles is covered by clone fragments. This means that a big portion of the whole launchfile is identical to another launchfile. For *(d)*, *(e)* and *(a)*, which are packages with high code cohesion of the clones, the duplicated code fragment may be shared with minor changes between several launchfiles. The

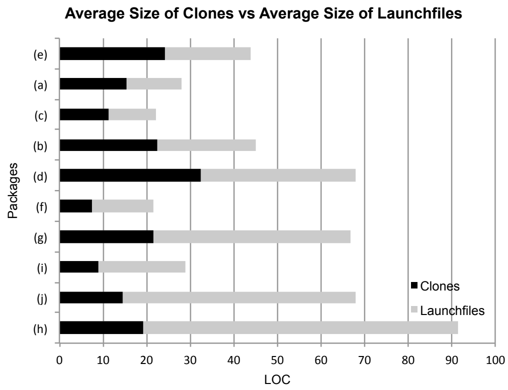

Fig. 4: Size of all clones and its launchfiles, ordered by proportion of cloned code.

other packages vary from 21% to 35% of cloned code, which is clearly a non-negligible portion.

C. What is Cloned?

After having detected and described the above, it is interesting to go deeper to know what kind of code is actually more often cloned. The ROS Launchfile XML format defines several tags, arguably the most common of which are:

- *Node*: Specifies a node to be launched and the package where it is defined.
- *Include*: Specifies the path to another launchfile whose tags will be imported.
- *Remap*: Allows to remap names, binding internal arguments defined in a node with others defined in the current launchfile.
- *Env*: Set values for environment variables that are valid under the scope of a node, launchfile or certain machine.
- *Param*: Defines a single parameter to be set in the pool of global parameters (available for all running nodes).
- *Rosparam*: Allows massive parameter definition by importing them from an external YAML-formatted file or export current parameters to a file.
- *Arg*: Permits to abstact certain variables, delegating the concrete value to be set afterwards. This tag makes launchfiles more abstract favoring reusability through *include*.

We counted all instances when the above tags were cloned for each clone pair belonging to our 10 prioritized packages, as shown in Figure 5.

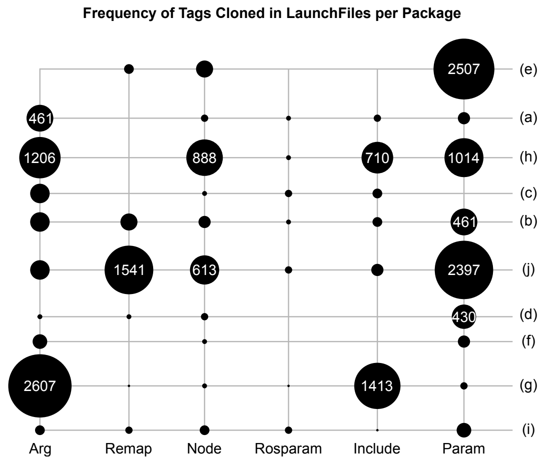

Fig. 5: Which tags are cloned more often per package

Among packages there is no clear pattern of tag clone frequency. For instance package *(j)* has many *Param* and *Remap* tag clones, however for *(g)*, *Include* and *Arg* are usually cloned. Package *(h)* presents a regular amount of clones in all tags but *Remap* and *Rosparam*

V. Related Work

This work is essentially a *Software Ecosystems* investigation on *Code Duplication* so we discuss related work in this order.

Software Ecosystems is an emerging area of Empirical Software Engineering that aims to understand how software projects interact and evolve with other projects in the same ecosystem. Using this point of view certain software development process aspects are studied in a group of related software projects. Studies cover aspects from revealing implicit dependencies between projects [10], [3] and reviewing explicit dependencies [12], [1] to analyzing developers' behaviour due to evolution in the ecosystem [15].

Code duplication is a more mature area compared to the two previously discussed. A large body of work exists that is focused on the analysis of a single project in terms of code duplication and its consequences on its development [16]. Many tools have been implemented for clone detection, and there is even work on comparative evaluation of these tools [2], as well as the creation of a clone detection benchmark suite [7].

VI. Conclusion and Future Work

In this work, we have performed a first analysis of the ecosystem around the ROS robotics middleware, with a focus on the presence of code clones in ROS launchfiles; configuration files that allow the setup of different robot processes at once.

We first provided a global overview of the ROS ecosystem: 1560 ROS packages with 47796 files totalling 1.73 GB. Notably, in this ecosystem code files only consist 39% of all files and the ecosystem has a high heterogeneity of files.

Considering the presence of launchfiles in this ecosystem, we see that only 34% of ROS packages (133) define launchfiles. Of these packages, half define only one and 30% define between 2 and 5 launchfiles. Considering all packages that contain more than one launchfile, 25% of these contain code clones. Focussing on these packages, almost half present one clone, 21% two or three clones, and 12% four to six. Focussing on packages with up to six clones, we find that there is a high similarity between the different clones: more than 45% on average. This shows that that developers simply reuse relevant code fragments in launchfiles by copy-and-pasting.

We performed an in-depth study of 10 representative packages according to their overlap and clone cohesion, *i.e.* having several launchfiles that share big portions of code where moreover this fragment is repeated in all of them with minor differences. It reveals that in these cases in general there is an excessively high amount of launchfiles with clones. Moreover there are two kinds of clones: either one big portion of a launchfile is a clone or one small clone fragment will be present in many launchfiles. And lastly, we do not find a clear pattern in the XML tags of the launchfiles that are typically cloned.

Our future work will consist of investigating the commit histories of these 10 representative packages to establish the processes that yielded the code clones to further understand the development scenarios that resulted in this situation. We will then investigate the creation of tools to detect code clones in launchfiles and propose the refactoring of them to reduce the amount of clones through the use of the include tag.

References

[1] P. Abate, R. Di Cosmo, L. Gesbert, F. Le Fessant, R. Treinen, and S. Zacchiroli. Mining component repositories for installability issues.

[2] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo. Comparison and evaluation of clone detection tools. *Software Engineering, IEEE Transactions on*, 33(9):577–591, 2007.

[3] K. Blincoe, F. Harrison, and D. Damian. Ecosystems in github and a method for ecosystem identification using reference coupling.

[4] S. Bragagnolo, L. Fabresse, J. Laval, P. Estefó, and N. Bouraqadi. Pharos: a ros client for the pharo language. http://car.mines-douai.fr/category/pharos/, 2014.

[5] J. H. Johnson. Identifying redundancy in source code using fingerprints. In *Proceedings of the 1993 conference of the Centre for Advanced Studies on Collaborative research: software engineering-Volume 1*, pages 171–183. IBM Press, 1993.

[6] D. Kohler. Rosjava. http://www.ros.org/wiki/rosjava, may 2012.

[7] A. Lakhotia, J. Li, A. Walenstein, and Y. Yang. Towards a clone detection benchmark suite and results archive. In *Program Comprehension, 2003. 11th IEEE International Workshop on*, pages 285–286. IEEE, 2003.

[8] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. Cp-miner: Finding copy-paste and related bugs in large-scale software code. *Software Engineering, IEEE Transactions on*, 32(3):176–192, 2006.

[9] M. Lungu, R. Robbes, and M. Lanza. Recovering inter-project dependencies in software ecosystems. *Conference on Automated Software Engineering*, pages 309–312, 2010.

[10] M. Lungu, R. Robbes, and M. Lanza. Recovering inter-project dependencies in software ecosystems. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*, pages 309–312. ACM, 2010.

[11] B. Marthi. Roslisp. http://wiki.ros.org/roslisp, june 2015.

[12] C. Mëlick, M. Tom, D. C. Roberto, and V. Jerome. A historical analysis of debian package incompatibilities. In *Proceedings of the 12th Working Conference on Mining Software Repositories*, 2015.

[13] A. Monden, D. Nakae, T. Kamiya, S.-i. Sato, and K.-i. Matsumoto. Software quality analysis by code clones in industrial legacy software. In *Software Metrics, 2002. Proceedings. Eighth IEEE Symposium on*, pages 87–94. IEEE, 2002.

[14] M. Quigley, K. Conley, B. P. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng. Ros: an open-source robot operating system. In *ICRA Workshop on Open Source Software*, 2009.

[15] R. Robbes, M. Lungu, and D. Röthlisberger. How do developers react to api deprecation?: the case of a smalltalk ecosystem. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, page 56. ACM, 2012.

[16] C. K. Roy and J. R. Cordy. A Survey on Software Clone Detection Research. *Queen's School of Computing TR*, 115:115, 2007.