Towards an Advanced ROS Package Generator

Anthony Remazeilles and Jon Azpiazu
TECNALIA, Paseo Mikeletegi, Parque Tecnológico, San Sebastian, Spain

Keywords: Robotics, Software Engineering, Code Generation.

Abstract: This paper describes a tool for generating ROS packages and nodes. Compared to the relatively basic traditional package creation method, this tool can generate a whole node structure, including its life-cycle and the exposed interface to other ROS nodes. Following a separation of concerns, the developer only defines the interaction means in a XML file, and the tool provides the whole skeleton of the nodes, including the interface creation and management. This way, the developer can focus on his real added value, the implementation of the node logic. Compared to advanced node management frameworks proposed in literature, the tool proposed does not require the developer to understand and agree on complex high-level architecture models. The developer only has to select a template model, and to provide the desired interface to get the code generated. The package generation is made possible thanks to package templates, and we provide with the generator tool two templates for creating nodes either in C++ or Python. The user has also the possibility to design his own template, so that he can develop the one that best fits his needs and best practices. The package generator code is accessible on public repository hosting facilities.

1 INTRODUCTION

The *Robot Operating System* (ROS) (Quigley et al., 2009) has become the standard framework for developing robotic solutions. Its popularity in the research community does not need to be demonstrated anymore and initiatives like ROS-Industrial promote its use in the context of industrial applications.

Surprisingly, in comparison to the outstanding number of applications developed with ROS, there is very little material for creating the basic component of a ROS development, named ROS package. A ROS package is usually created using terminal commands such as `catkin_create_pkg` which enables introducing meta information about the package to generate (list of dependencies, author email, etc). Once executed, a folder is created containing the files `package.xml` and `CMakelists.txt` that are partially filled out according to the command arguments. These two files are the basics that define a ROS package. Then, the component creation is left entirely to the developer. On one hand, this is positive since the developer has a complete freedom for the implementation. On the other hand, we can highlight the following drawbacks:

Loss of Time: Every time a new package or node is created, the developer has to do it from scratch. Clever copy and paste may be used to get inspiration from previously designed nodes but this is usually error-prone. The developer is very likely to loose time in re-implementing the basic layers of the node.

Implementation Quality: There is a lot of documentation proposing good practices on packages and nodes structure. Nevertheless, their use depends on the developer, according to his programming knowledge and expertise, his time constraint or motivation. Thus, for any of these reasons, it is probable that a developer may take some inappropriate implementation decisions. Also, a developer team may have some preferred implementation schemes (documentation style, header contents, communication models, ...). But there is no simple solution to generate packages or node skeletons from these guidelines.

Node Life-cycle Hidden in Source Code: The life-cycle of a node refers to its different stages of execution (a more detailed description is given later). This information is crucial to understand how to use the node and interact with it, and should be detailed in the documentation. It is thus up to the developer to provide a good and updated description. Otherwise, it may be needed to dig into the code to figure out how the node behaves, which can be very time-consuming.

Node Interface Hidden in Source Code: The node interface (communication mechanisms exposed to other ROS nodes) is usually described in the node's

Remazeilles, A. and Azpiazu, J.
Towards an Advanced ROS Package Generator.
DOI: 10.5220/0007834002430250
In *Proceedings of the 16th International Conference on Informatics in Control, Automation and Robotics (ICINCO 2019)*, pages 243-250
ISBN: 978-989-758-380-3

documentation. If it is not documented, an alternative is to run the node, and use ROS introspection tools to figure it out. Nevertheless, it may not be evident to infer the complete interface this way.

Code Evolution and Reuse Potentially Complex: The two previous points highlighted the potential difficulties for users to understand how to use an existing node. The evolution of the code (by other developer, but even by the author) may also be complex if the current implementation is not well described and if it does not follow best practices. Even little extensions of the functionality or interface may lead to significant development time for (i) understanding the code, and (ii) reshaping the structure to follow better implementation strategies.

Based on this analysis, we propose a novel ROS package generator to speed up and ease ROS development[1] The package generator relies on package templates, but any advanced developer can implement his own one, based on his needs and programming policies. For the user, the package generation only requires the selection of the template, and a specification file describing the interface of the package. The package generator then automatically provides a whole package by adjusting the template according to the specifications. The component skeleton obtained enables the developer to focus on his real contribution, that is the node logic implementation. The tool is also provided with update mechanisms enabling the developer to update the interface of the package generated without loosing the code already inserted.

Next Section presents the literature related to ROS packages and nodes creation. Section 3 describes the component we have developed, and in Section 4 we show how it can be used with the templates provided, and how new templates can be created. Conclusions and future work are mentioned in the last Section.

2 STATE OF THE ART

Several high-level tools exist for automating the creation of ROS nodes and packages. They reduce the cost of code production by using hidden code templates or skeletons. If the proposed template fits his needs, the developer can focus on the logic implementation and let the automation tool prepare the rest of the architecture.

ROSLab is a high-level programming language (Bezzo et al., 2014) that is an extra layer added on the top of ROS to simplify the lower level code generation. In a Java-based graphical interface it enables to connect nodes through their communication interface to create a complete application. The underlying code generation uses ROSGen component implemented in Coq (Meng et al., 2015). Unfortunately, that solution has not been maintained since 2017.

ROSMOD (Robot Operating System Model-driven development tool suite) also provides graphical tools for rapid prototyping and deploying large-scale applications (Kumar et al., 2016). It follows a component-based approach structure, and is said to be a refinement of the ROS component model. ROSMOD intends to reduce the amount of time and effort developers spend installing, configuring, and maintaining applications. Nevertheless, it requires agreeing with the proposed component model, that is slightly different from the traditional ROS one, which may be acknowledged only by advanced developers.

BRIDE (*BRICS Integrated Development Environment*) is one the main outcomes of the European project BRICS (Bischoff et al., 2010; Bubeck et al., 2014). Following a Model Driven Engineering approach, it provides an abstract representation of component interfaces and behaviors, as well as an automatic model validation and code generation (in ROS or Orocos). BRIDE is integrated as an Eclipse plugin, so that the developer can graphically design nodes and their communication interface. The development is following the spirit of Component-Based Software Engineering, targeting quality, technical and functional reusability (Brugali and Shakhimardanov, 2010). Considering that a software component is defined to be a unit of composition with contractually specified interfaces and explicit context dependencies only, BRICS stresses the clear distinction in between the interface (framework specific) and the implementation of the component functionalities (framework independent). From the definition of a component interface, BRICS prepares the ROS node structure and the communication tools, and places in a separate file the skeleton of the code to be filled by the user. The concepts followed by BRICS are of major importance for developing stable components with clear interfaces. Unfortunately, the developments have stopped since 2017 and at the ROS Indigo release. Furthermore, changing the life-cycle or the generic structure of the ROS component pattern requires strong expertise in Java programming and Eclipse plugin development.

So far, it is not evident whether any of these solutions has been broadly accepted and / or used. We see two limitations that could explain such lack of community acceptance. First of all, these tools rely on meta models of software architecture placed on the top of ROS ecosystem. Even if their technical sound-

[1]See Repository https://github.com/tecnalia-advanced manufacturing-robotics/ros_pkg_genros_pkg_gen in github

ness and quality may be relevant, they nevertheless require a (too) significant effort from the developers to learn, and thus to be willing to use these solutions. Our package generator only requires selecting a template based on ROS concepts (like the life-cycle) and filling a XML file defining the desired interface. Secondly, the maintenance of some of these software unfortunately stopped with the closure of the projects they came from. To maintain these developments, a significant effort to understand the framework implementation is required since the component patterns are usually defined implicitly in the source code.

The package templates used by our generator are not embedded in the generator code, and new templates can be created following instructions similar to templating languages, easing the adaptation of the patterns to developer needs. Also the package generator is implemented in Python, which is a common programming language within the ROS community.

As already stated, it is very important to define explicitly the interface of a node. But it is also crucial making clear what is its life-cycle. Even though part of it may be inferred from the interface definition, critical aspects may not be easily inferred , such as when the node computation starts, can (should) we stop and resume the node activity during the application, and so on. A particular care is taken with such matters in ROS2, through the concept of managed nodes (ROS, 2018). Managed nodes implement a state machine indicating whether the node is unconfigured (just instantiated), inactive (configured but not running), active (performing its computation) or finalized (before destruction). The definition of a managed node requires implementing the different transitions from one state to another. By structuring the node's life-cycle, the use of managed nodes is less dependent on the developer's implementation choices. This should definitely ease monitoring applications and the reuse of components. Nevertheless, the duty of creating nodes and defining the communication interface remains on the user's side, so that a package generator tool would still be of interest.

3 THE PACKAGE GENERATOR

Our package generator leads to the following workflow (from the source folder of a ROS workspace):

```
# definition of the package spec in a file
$ gedit mygreatpackage.ros_package
$ rosrun package_generator generate_package \
mygreatpackage.ros_package python_model_update
```

mygreatpackage.ros_package is a XML description of the package specification, including all nodes to be created and their desired respective interfaces (an example of such file is provided on Fig. 1). python_model_update is the name of the template to be used to generate the new package.

Once executed, a complete skeleton of the package and included nodes is generated automatically, including the definition and implementation of the interface as well as the nodes' life cycle, according to the package model selected. The package documentation with all nodes specification (interface, node life-cycle) is also automatically generated. The developer just has to write the node intelligence, in well designated specific regions. If he decides to change the interface (for example adding a topic subscription), the XML description file is updated and the previous command relaunched. The user contribution in the tagged areas is kept, as well as additional files not present in the template if the developer explicitly requires it. The Developer still can change the implementation anywhere, diverging from the original node pattern. In such case, the update functionality could not be used anymore.

Next Section provides some terminology and Section 3.2 gathers the requirements that guided the tool design. Then Section 3.3 details the needed information to launch the package generation, while Section 3.4 describes the package generator implementation.

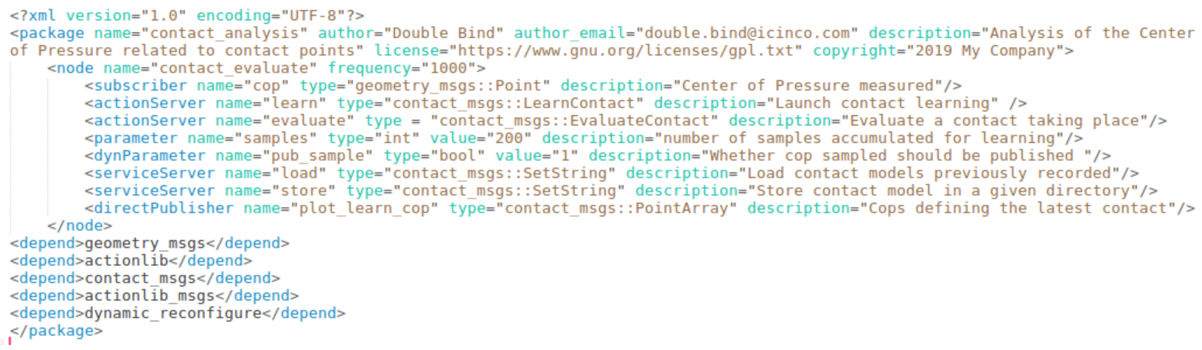
3.1 Package Generation Terminology

We start with the definition of different concepts we will frequently use.

Template Package: set of files, in any of the ROS known languages, constituting a package skeleton (like python_model_update in the previous example). Each file of the template is composed of (i) code to be reproduced as is in the generated file, (ii) tags to be interpreted by the code generator and (iii) specific areas indicating where the Developer should insert the node's intelligence.

Template Designer: person developing a template package. He is in charge of deciding how the XML specification file is affecting the code skeleton, and where the Developer should provide the node logic. The Template Designer provides the whole package and node pattern, from the node interface to its life-cycle, and can also automate the generation of the documentation. He can also restrict the possible interface (by implementing only a subset of the ROS communication tools). It requires a deeper understanding of the package_generator mechanism.

Package Developer: person who wants to create a new ROS package. His responsibility is (i) to select the suitable template, (ii) to define the package specification accordingly to the template documentation,



```xml
<?xml version="1.0" encoding="UTF-8"?>
<package name="contact_analysis" author="Double Bind" author_email="double.bind@icinco.com" description="Analysis of the Center
of Pressure related to contact points" license="https://www.gnu.org/licenses/gpl.txt" copyright="2019 My Company">
    <node name="contact_evaluate" frequency="1000">
        <subscriber name="cop" type="geometry_msgs::Point" description="Center of Pressure measured"/>
        <actionServer name="learn" type="contact_msgs::LearnContact" description="Launch contact learning" />
        <actionServer name="evaluate" type = "contact_msgs::EvaluateContact" description="Evaluate a contact taking place"/>
        <parameter name="samples" type="int" value="200" description="number of samples accumulated for learning"/>
        <dynParameter name="pub_sample" type="bool" value="1" description="Whether cop sampled should be published "/>
        <serviceServer name="load" type="contact_msgs::SetString" description="Load contact models previously recorded"/>
        <serviceServer name="store" type="contact_msgs::SetString" description="Store contact model in a given directory"/>
        <directPublisher name="plot_learn_cop" type="contact_msgs::PointArray" description="Cops defining the latest contact"/>
    </node>
<depend>geometry_msgs</depend>
<depend>actionlib</depend>
<depend>contact_msgs</depend>
<depend>actionlib_msgs</depend>
<depend>dynamic_reconfigure</depend>
</package>
```

Figure 1: XML Package specification file example.

and, once the package is created, (iii), to add the node logic to the generated code.

Node Logic: code added by the Developer to the generated package. If we assume that the package template takes care of the node interface and its life-cycle, the node logic should only consider the implementation of the node computation.

Node Interface: list of communication means a node is proposing to the rest of the ROS nodes. It is based on all standard ROS tools, i.e. topics, services, actions, parameter and dynamic parameters, tf. Ideally, the template should be designed to make a clear separation between the interface management (like subscriber creation and callback), and the exchanged information use or generation (like processing a message received to produce a message to send). Also the template should handle the interface description in the documentation file (Readme.md file).

Node Life-cycle: behavior of the node from its creation to its termination. Every ROS node has a life-cycle, although it may not be clearly described (and implemented). The Template Designer is ideally responsible for defining the node life-cycle, through the proposed template. He should therefore mention it in the template description, and handle it in the node skeleton. This way, a Developer knows by selecting a template how the life-cycle will be, and, once a package is generated and filled by the Developer, a user of the package will have a clear description of the component's behavior.

It is clear that a strong responsibility is placed on the Designer's shoulders. Indeed, the quality of the template, the node life-cycle and interface implementation and description strongly rely on him.

3.2 Package Generator Objectives

The package generator has been implemented for addressing the following objectives:

Automatic Code Generation, including node life-cycle pattern and node interface: after the package generation, the Developer focuses on the node logic.

Node Creation based on the Interface: the Developer should only define the expected input and output information to create a node. In the templates provided, the XML configuration mainly focuses on the interface. Code related to the interface creation and management is then automatically created.

Separation of Concerns: this is related to a clear distinction of the interface definition and management and its implementation. This separation is totally dependent on the templates, not on the code generation. The templates proposed follow this leitmotiv, by explicitly using different files for the two aspects.

Adjustability/customizability: the template model is not hard-coded in the package generator, and new ones can be added. We provide right now two templates, one for C++ nodes and another for Python nodes. A Template Designer could use these models as example to produce templates more suitable to his need. Template creation enables teams to agree and follow common patterns. Also, by using code generation tools instead of inheritance mechanisms, the Developer has access to all the code in his package, and can, if he sees the needs, change any part of the generated code.

Keep It Simple: the use of the package generator is made simple to ease its adoption by the community.

3.3 User Input Information

The generation of a package requires two information: (i) a specification file and (ii) the name of the selected template. The specification file is a XML file which structure is strongly inspired by the BRIDE model. An example of specification file is presented in figure 1. The package attributes, including its name, are meta-data associated to the package, used to fill the package.xml file and the documentation.

The package tag contains a node tag per node to be generated (so far all nodes follow the same pattern). The attributes of a node provide specifications for it. In the current templates, the Developer can specify the node name and the update frequency.

Inside the node tag is then described the node's interface. In the provided templates, the Developer has access to all standard ROS communication means, i.e publishers, subscribers, parameters and dynamic parameters, service clients and servers, action clients and servers, tf listeners and broadcasters.

Most of the interface components are described by the same attributes, i.e. name, type and description. The first two attributes are required for the automatic generation of the interface code, while the latter is needed for the documentation generation.

Finally, a set of dependencies are provided. Current templates automatically add these packages to the files CMakeLists.txt and package.xml. When the XML file is loaded, we also check the packages used by the interface, so that additional dependencies may be automatically added accordingly.

3.4 Package Generator Details

The package generator, implemented in Python, is composed of the following files (see Fig. 2):

- package_generator.py: orchestrates the whole package generation, given a template directory, and a XML package description.
- package_xml_parser.py: is responsible for parsing the XML package description.
- code_generator.py: generates a file from a XML node description and a file template.
- update_mgt.py: provides the update functionality for packages already created.

The script package_generator.py orchestrates the whole package generation, following the package template. A package template is a regular directory which typical contents are presented on figure 3. It must contain two folders: config and template. The first one provides configuration information, while the second gathers the skeleton of all files to be generated.

In the configuration folder, the file dictionary.yaml defines the tags a Developer can use in his XML configuration file. The Designer defines here each possible interface, with the related attributes. In the example in figure 4, the provided interfaces are very similar to the standard ROS ones, but the Designer can also add new concepts, or even reduce the number of interfaces allowed if appropriate. Note that, on the Developer side, a tool is provided to generate a XML specification skeleton with all the accepted interfaces by the template. Finally, the file functions.py enables

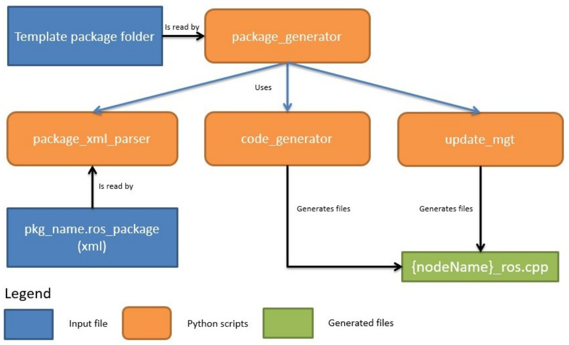
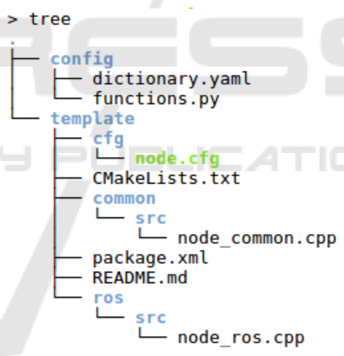


Figure 2: Main components of the package generator.

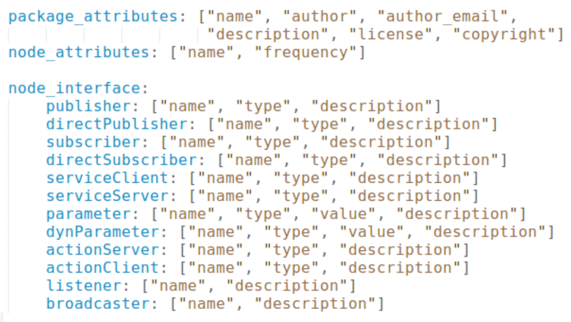
the Designer to add simple functions that can be used during the code generation (mentioned later on).

The global script package_generator.py invokes the script package_xml_parser to load and pre-process the package description, and then handles the generation of all template files by calling the script code_generator.py. If the XML configuration file contains several nodes, files containing node in their name (node_common.cpp and node_ros.cpp in the example on Figure 3) are generated for each node specified, while the other files are only generated once.



```
> tree
.
├── config
│   ├── dictionary.yaml
│   └── functions.py
└── template
    ├── cfg
    │   └── node.cfg
    ├── CMakeLists.txt
    ├── common
    │   └── src
    │       └── node_common.cpp
    ├── package.xml
    ├── README.md
    └── ros
        └── src
            └── node_ros.cpp
```

Figure 3: Typical contents of a package template.



```
package_attributes: ["name", "author", "author_email",
                     "description", "license", "copyright"]
node_attributes: ["name", "frequency"]

node_interface:
    publisher: ["name", "type", "description"]
    directPublisher: ["name", "type", "description"]
    subscriber: ["name", "type", "description"]
    directSubscriber: ["name", "type", "description"]
    serviceClient: ["name", "type", "description"]
    serviceServer: ["name", "type", "description"]
    parameter: ["name", "type", "value", "description"]
    dynParameter: ["name", "type", "value", "description"]
    actionServer: ["name", "type", "description"]
    actionClient: ["name", "type", "description"]
    listener: ["name", "description"]
    broadcaster: ["name", "description"]
```

Figure 4: Specification of the dictionary handled by a given package template (file dictionary.yaml).

The script code_generator.py requires a template file, the specifications of the global package as well as the specifications of one node. The template file is scanned, looking for specific markers. Currently, for simplicity, the markers are all of format {instruction}. The instructions are purely related to the different tags the Developer can introduce in his XML file. More exactly, for all interface types, the Designer is provided with the generator instructions {ifinterface} and {forallinterface}. As an example, let us assume that a template file contains the following code:

```
{ifpublisher}
// defining all publishers
{endifpublisher}
{forallpublisher}
ros::Publisher {name}_ = n_.advertise<{type}>("{name}", 1);
{endforallpublisher}
```

The comment line will only be generated if at least one publisher is defined. The publisher definition will be repeated for all the publishers defined, using the values of the attributes name and type found in the XML description of this node.

The script code_generator.py also handles instructions of type {apply-func}. It will apply the function func with the interface attributes as input parameters. For example, the type of an interface is provided in the XML file with the format PackageName::Type, i.e as we would use it in regular C++ code. Dedicated functions are introduced for mapping this type to Python formats, for extracting the package, the type, etc. All these functions are defined in the configuration file functions.py provided with the package template.

If the Developer executes again the package generation after its creation, the package_generator detects the presence of the package already created, and assumes that an update of that package is requested. A copy of the current package is placed in a temporal folder, and all files are then generated, and compared to their previous version. More exactly, the script update_mgt.py focuses on the areas delimited as follows:

```
# protected region user update begin #
rospy.loginfo("Update Received value: {}".format(data.
    in_counter))
# protected region user update end #
```

All Developer's code inserted between these tags is indexed and reinserted in the new code generated. That way the code already inserted by the Developer is maintained upon update.

4 USE AND LESSONS LEARNED

As an illustration of use of the package generator, we have implemented two package templates. Both provide a standard node template with a clear life-cycle that can already handle a large variety of developments. They can also help Template Designers implementing their own models.

Next Section describes their implementation, while Section 4.2 provides insight on the lessons learned during the template design and use for real packages. Finally we will compare the tool outcomes to the objectives initially defined in Section 3.2.

4.1 Templates Provided

The two templates provided are similar in term of behavior. They enable generating packages with nodes respectively in C++ and Python. We focus on the C++ version, as most of the description holds for both.

Following the separation of concerns strategy, we distinguish the node communication and coordination from the computational part. The Developer's contribution is only expected in the latter, all other aspects are automatically generated. In terms of life-cycle pattern, the proposed scheme is a periodical update, in which at each iteration of the main loop, the computational layer gets the latest messages received (through message subscription) to generate the related output to be transmitted (through message publication).

The separation of concerns is materialized by the presence of two files per node:

- ros/src/[node_name]_ros.cpp ROS interface and life-cycle implementation. It defines the class Ros[NodeName].
- common/src/[node_name]_common.cpp will contain the node logic provided by the Developer, mainly by filling a class named [NodeName]Impl.

The class Ros[NodeName] defines the ROS communication interface. It contains also attributes related to classes defined in [node_name]_common.cpp:

- [NodeName]Config: contains the (dynamically adjusted or not) parameters.
- [NodeName]Data: constains the input messages received and the output messages to be sent.
- [NodeName]Impl: will contain the Developer implementation of the node.

File [node_name]_ros.cpp contains the main function. After creating an instance of Ros[NodeName] and a configuration step an infinite loop is started. All messages received are stored in an object of type [NodeName]Data. At each iteration, the method update of the class NodeNameImpl is called:

```
void NodeImpl::update([NodeName]Data &data, [NodeName]
    Config config)
```

The parameter data contains the latest messages received through subscription. Based on this input, the method update implemented by the Developer can prepare messages to be sent and store them in specific attributes of the variable data. The publication is handled by the instance of the class Ros[NodeName] at the return of this update method. The parameter config contains all the application parameters (from the parameter server, or handled through dynamic reconfigure). This way, in a classical publisher / subscriber scheme, this update function provides to the Developer all needed input to prepare the output.

The Developer's contributions are only expected within the file [node_name]_common.cpp, in locations specified with the user contribution tag. The Developer may still update the code elsewhere but the update mechanism will not keep these changes then.

In the file [node_name]_common.cpp, the class [NodeName]Passthrough gathers components violating the interface / implementation separation paradigm. We define in it all interface components that may be directly accessed from the Developer's side. Furthermore, in the [NodeName]Data class, all input messages are associated with a boolean flag indicating whether the data has been updated since last update call. The Developer can use such information for the preparation of the output material. Similarly, a boolean flag is associated to all output messages, so that the Developer can inform whether it is necessary to publish any data after the update completion.

An extensive documentation file template is provided, to get a complete description of all the communication means provided by each node. It is automatically generated from the XML description file, and specific user contribution areas are prepared to enable the Developer extending the information.

Finally the ROS files related to the build operation, package.xml and CMakelists.txt, are also contemplated in the package template, and automatically filled from the XML description. A package freshly created is thus already ready to be built, without modifying any of the generated files.

4.2 Lessons Learned

We are using the package generator in several running projects. In Python for example the code generated can represent around 80% of the total node code, and a quite complete interface description is directly extracted from the XML specification.

The proposed templates are usually sufficient to handle most of the packages needs, but we also generated more dedicated templates to better represent some specific models, such as pure service or action nodes, pure messages and services definition packages, filter nodes, ... We usually start with the default template and then decide whether the adaptations we feel necessary should be added to the default model or are worth the creation of a new template.

The update mechanism has also shown to be efficient for updating already created packages after an adjustment of the template itself. This occurred for instance when we inserted the flags for signalizing the update of input messages or for signaling if output messages should be published. All packages created before could be directly upgraded with such functionality by re-executing the package generator.

The capability of defining ad-hoc interface attributes has also enabled to characterize and formalize different communication models. For instance, in several projects we have seen that waiting the update period for processing or publishing messages may be restrictive. We have thus decided to implement a dedicated interface, named directPublisher and directSubscriber. These message communication managers are regular ROS publishers and providers that are handled out of the update mechanism (within class [NodeName]Passthrough), for specific interactions that cannot wait for an update cycle.

Finally we observed that when using such template mechanism, the Developer has to think of the component he wants to create before starting implementing it. Which template and related life-cycle best fits to a given task? What is the targeted interface? These questions have to be addressed yet from the beginning, and the update mechanism always gives a chance to adjust the decision initially taken.

4.3 Completion of Initial Objectives

We highlighted in Section 3.2 five objectives. The automatic code generation is definitely achieved. Once the package generation is launched, the Developer only has then to insert the node logic. All the ROS connectivity is automatically generated. There is a significant improvement with respect to the standard catkin_create_package command.

The generation of the node life cycle, the node interface, as well as the node creation based on the interface is effectively happening with the package templates we provide. We also explicitly separate the ROS interface and node cycle from the node intelligence by placing them within different files.

Nevertheless, the completion of these objectives totally relies on the template provided by the Designer. On one hand we can argue that the Designer may then define a template that violates these objectives. On the other hand, this gives total freedom to the Designer for defining the template that matches exactly the need of his team.

The possibility of defining new templates is also demonstrated as we already propose two different templates for two different languages, C++ and Python. The Designer can define not only his own template, but also the interface itself, with the related tags and attributes the Developer should use in the XML file.

With respect to the last objective, *keep it simple*, it is achieved on the Developer side considering the limited information requested to trigger the code generation: the appropriate template, and the targeted node interface. The Template Designer may have a more complex job, for defining the appropriate template. Nevertheless this has to be counterbalanced with the saved time by reducing the effort needed for code maintenance, refactoring etc., that is likely to happen otherwise, even if the team has a code policy.

To finish, we consider the possibility to change the generated code as a positive point. The generated code is not using advanced meta model that may be complex for the Developer to tweak if the situation requires it. Our generated code can still be changed by the Developer, even though this is not preferable.

5 CONCLUSIONS

We described a ROS package generator that generates complete package and node code based on a given template and a simple XML description of the desired interface. This tool is not restricted to a unique package template, and Designers have the possibility to implement new templates. The template creation is relatively simple, and uses instructions automatically adjusted to the template characteristics. Also we provide an update mechanism so that a Developer can adjust or extend the proposed interface, without loosing the node logic previously introduced.

Several extensions are envisioned to enhance the capabilities of the package generator. One of them is to enable the generation of other ROS components. Currently only ROS nodes are created, which is enforced by the use of the special tag node in the XML dictionary. By enabling other types of tags, we believe the code generator could easily be extended to other structures, such as ROS controllers.

The code generator started with a simple but efficient in-house templating language derived from the interface parameters. We are strongly considering more mature code generator tools, such as Jinja, that would give access to more complex code generation scheme in the package templates.

We are also considering migrating the generation layer, currently at the level of the package, to a lower layer, at node level, or more generally speaking at component level. That would enable mixing different templates in a given ROS package, like for example a ROS node in C++, another one in Python ...

It would be interesting porting the package generator to ROS2. The package code is almost pure Python, using quite limited ROS functionality. Such migration should be quite straightforward.

Finally, it would be an added value to provide related plugins for some code editors. This would help the writing of the package configuration file, providing appropriate text completion to the Developer.

ACKNOWLEDGEMENTS

Supported by the Elkartek MALGUROB and the ROSIN project under the European Union's Horizon 2020 research & innovation programme, grant agreement No. 732287. The authors would like to thank Dr. Andrzej Wasowski for the helpful suggestions regarding the structure and contents of this article.

REFERENCES

Bezzo, N., Park, J., King, A., Gebhard, P., Ivanov, R., and Lee, I. (2014). Demo abstract: ROSLab — A modular programming environment for robotic applications. In *ACM/IEEE ICCPS*, pages 214–214, Berlin, Germany.

Bischoff, R., Guhl, T., Prassler, E., Nowak, W., Kraetzschmar, G., Bruyninckx, H., Soetens, P., Haegele, M., Pott, A., Breedveld, P., Broenink, J., Brugali, D., and Tomatis, N. (2010). BRICS - Best practice in robotics. In *ISR / ROBOTIK*, pages 1–8.

Brugali, D. and Shakhimardanov, A. (2010). Component-Based Robotic Engineering (Part II). *IEEE Robotics Automation Magazine*, 17(1):100–112.

Bubeck, A., Weisshardt, F., and Verl, A. (2014). BRIDE - A toolchain for framework-independent development of industrial service robot applications. In *ISR/Robotik*, pages 1–6, Munich, Germany.

Kumar, P. S., Emfinger, W., Karsai, G., Watkins, D., Gasser, B., and Anilkumar, A. (2016). ROSMOD: A Toolsuite for Modeling, Generating, Deploying, and Managing Distributed Real-time Component-based Software using ROS. *Electronics*, 5(3).

Meng, W., Park, J., Sokolsky, O., Weirich, S., and Lee, I. (2015). Verified ROS-Based Deployment of Platform-Independent Control Systems. In *NASA Formal Methods*, volume 9058, pages 248–262.

Quigley, M., Conley, K., Gerkey, B., Faust, J., Foote, T., Leibs, J., Wheeler, R., and Ng, A. (2009). ROS: an open-source Robot Operating System. In *ICRA Workshop on Open Source Software*.

ROS (2018). ROS2 overview - Managed Nodes. https://index.ros.org/doc/ros2/Managed-Nodes/. Accessed: 2018-11-22.