Journal of Software Engineering for Robotics 8(1), December 2017, 3-16
ISSN: 2035-3928

Modeling Robotics Software Architectures with Modular Model Transformations

Kai ADAM Katrin HÖLLDOBLER Bernhard RUMPE Andreas WORTMANN
Software Engineering, RWTH Aachen University, Germany, www.se-rwth.de

Abstract—Robotics started employing architecture description languages (ADLs) to model software architectures. While facilitating software engineering, this introduces a gap when reusing solutions encoded in middleware modules. Existing robotics architecture modeling focuses on domain challenges instead of tool modularity. Thus customizing robotics architecture modeling tool to generate solutions conforming to a different middleware is challenging. This could lead to a multitude of incompatible 'vendor-locked' tool modeling chains and hamper reuse in robotics software engineering. We propose a modular architecture modeling method that rests on the separation of model processing, model transformation, and code generation. This facilitates translating architecture models into modules compatible to the middleware of choice. We present this method for the component & connector ADL MontiArcAutomaton, which yields and extensible tool chain to translate software architecture models gradually into middleware modules. Using modular tool chains to support architecture modeling enables reaping the benefits of ADLs while reusing solutions encoded in popular middlewares and, ultimately, facilitates robotics software engineering.

Index Terms—Model-Driven Development, Architecture Description Languages, Modular Model Transformations, Code Generation

1 Introduction

Robotics is the most challenging domain for software engineering: successfully deploying even simple robotics applications demands expertise from various domains and integration of heterogeneous software modules. Robotics successfully has adopted [1] model-driven development (MDD) [2], [3], which facilitates integrating domain experts by lifting better comprehensible, abstract models to primary development artifacts. Parallel to their efforts, there is a large corpus of robotics solutions encoded in general-purpose programming language (GPL) artifacts that are specific to various robotics middlewares (such as Orocos [4], CLARAty [5], or ROS [6]). These are hardly accessible by MDD tools.

At the same time, the reuse promised by component-based software engineering [7] has been deemed crucial to reusable robotics architectures [8]. Similar to avionics [9] and automotive [10], robotics-specific architecture description languages (ADLs) [11], [12] lift the notion of components to *component models* (*cf.* BRICS [13], C-Forge [14], DiaSpec [15], SmartSoft [16], or V3CMM [17]).

Interfacing component models with the expertise encoded in the middleware-specific implementations is a prerequisite for efficient architecture modeling in robotics. However, many ADLs focusing on robotics are tied to hardly extensible MDD tool chains (including parsers, editors, code generators, etc.). Thus transforming the architectures' component models into artifacts compatible to a specific middleware unforeseen by the tool chain is challenging.

Based on experiences in software architecture modeling for automotive [18], cloud systems [19], and robotics [20], we present an extensible architecture modeling method that employs modular model-to-model (M2M) and model-to-text (M2T) transformations to enable transforming robotic software architecture models into implementations for arbitrary target platforms. To this effect, this method separates the concerns of architecture modelers from the concerns of model transformation developers and code generator engineers as depicted in Figure 1.

The architecture designer knows the employed ADL and creates a logical software architecture as required for the application under development. The model processing infrastructure parses the model and creates its internal representation. If the architecture model contains elements not easily translatable into target middleware modules (*e.g.,* hierarchies or complex

Regular paper – *Manuscript received July 15, 2017; revised November 15, 2017. Digital Object Identifier: 10.6092/JOSER_2017_08_01_p3*

- *This research has partly received funding from the German Federal Ministry for Education and Research under grant no. 01IS16043P. The responsibility for the content of this publication is with the authors.*

[AHRW17b] K. Adam, K. Hölldobler, B. Rumpe, A. Wortmann:
Modeling Robotics Software Architectures with Modular Model Transformations.
In: Journal of Software Engineering for Robotics (JOSER), 8(1):3–16, 2017.
www.se-rwth.de/publications

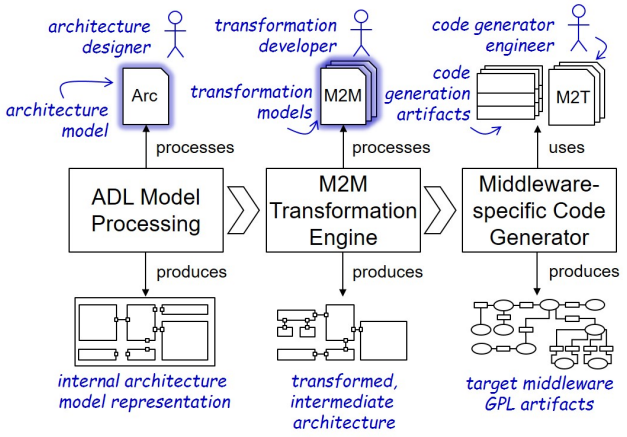

Fig. 1: Quintessential components, artifacts, and roles for pervasive, modular architecture modeling for different target middlewares illustrated on an example of the translation to ROS [6].

data types), the transformation developer provides appropriate model transformations. The M2M transformation engine parses the transformation models and applies these to the architecture. Ultimately, a middleware-specific code generator, provided by a code generator engineer with expertise in M2T transformations and target middleware, processes the transformed models and produces middleware-specific GPL artifacts.

This separation enables translating architectures into various intermediate representations better amenable to analysis or code generation. It also enables reusing architectures with different code generators, middlewares, and GPLs. While the MontiArcAutomaton infrastructure has been detailed in [20], [21], this paper focuses on its modular development method and its constituents: (1) modular, domain-specific model-to-model transformations; (2) template-based code generation; and (3) a case study for the translation of hierarchical MontiArcAutomaton architectures to ROS [6].

This paper is an extension of the work presented in [22]. It details the already presented transformations, presents additional M2M transformations, and explains the M2T transformation infrastructure. To this end, Section 2 motivates the benefits of modular architecture modeling by example, before Section 3 presents preliminaries. Afterwards, Section 4 describes the modular M2M transformations and Section 5 describes M2T transformation to ROS. Section 6 discusses observations and related work. Section 7 concludes.

2 Example

Consider a company producing the software architecture for a cleaning robot with two arms as depicted in Figure 2 (top). The architecture `CleaningRobot` comprises components providing functionality of various sensors, actuators, as well as pure software components. The actual functionality of two arms is not realized in the architecture but can be easily implemented by reusing existing middleware modules. The components are either hierarchically composed (*e.g.,* `Localization`) or atomic (*e.g.,* `Controller`) and the ADL distinguishes component types (*e.g.,* `Navigation`) and their instances (*e.g.,* `nav`). Components exchange messages via unidirectional connectors connected to their stable interfaces of typed, directed ports only. Ultimately, the architecture should be translated to (1) artifacts compatible to the Python client implementation of the robot operation system ROS [6] for execution; and (2) to Java for simulation [23]. For the latter, the company already has a black-box code generator. However, to ease comprehension and modeling, the ADL supports hierarchical components, whereas neither the Java simulator, nor ROS support hierarchies. As no modeling tool chain supports these transformations off-the-shelf, the company must develop appropriate transformations. To avoid implementing the elimination of hierarchies as pre-processing for the Java code generator and as part of the ROS code generator, this should be performed prior to code generation. For this, they desire to include appropriate M2M transformations. Moreover, this separation also enables to reuse existing ADL tooling (such as well-formedness checking or visualization) with the transformed architectures as well. After defining the corresponding M2M transformation, translation to Java and ROS requires less complex M2T transformations. They can easily realize on top of the FreeMarker[1] template engine and MontiArcAutomaton's code generation framework [20]. The resulting ROS nodes can easily interface with existing ROS nodes to reuse the encoded expertise.

Figure 2 depicts the results of both transformation activities: First the M2M transformation (1) eliminates the hierarchical components `CleaningRobot` and `Localization` and reconfigures the connectors accordingly. The result again is a valid architecture model that can be processed by existing tooling without modifications. Afterwards, the M2T transformation (2) translates the remaining components into ROS nodes and the connectors into individual topics. After translating the architecture to ROS nodes, adding existing ROS nodes to interface with the prepared topics is straightforward. This separation enables architecture developers to use the ADL of choice and connect the generated implementations to any target middleware. It also liberates the code generator developers from dealing with transformation challenges that (a) are either common to multiple translations or (b) are better expressible as M2M transformations.

3 Preliminaries

The presented infrastructure for modular model-driven development of robotics architectures relies on the MontiArcAutomaton C&C ADL, its code generation infrastructure,

1. http://freemarker.org/

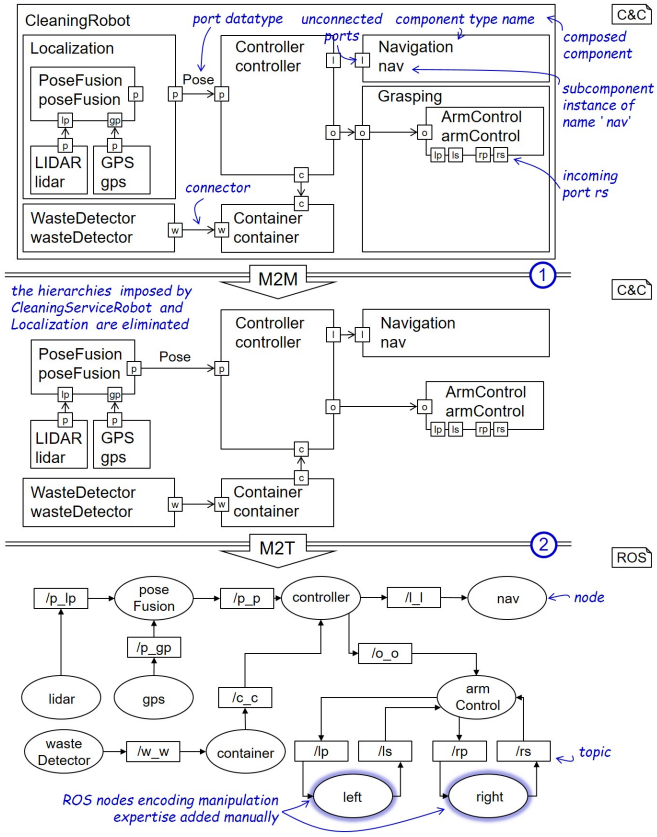

Fig. 2: Component & connector software architecture of a cleaning robot with two arms (top), after applying the M2M transformations for hierarchy elimination (middle), and after applying the M2T transformations producing ROS nodes and topics (bottom).

and the MATrans transformation language generated from MontiArcAutomaton. All of this is built with the MontiCore language workbench.

3.1 The MontiCore Language Workbench

MontiCore [24], [25] is a language workbench [26] for the efficient engineering of modular modeling languages. It comprises a metamodeling language to define languages, a compositional model checking framework, and a powerful code generation infrastructure. The metamodeling language is an EBNF-like, context-free grammar (CFG) that enables integrated definition of concrete and abstract syntax [24]. To validate constraints that are not expressible with CFGs, MontiCore features the compositional context condition framework [25]. The context conditions are well-formedness rules developed in Java, which yields a greater flexibility than employing specific constraint languages (such as the object constraint language (OCL)[27]). From the CFG of a modeling language, MontiCore generates the corresponding abstract syntax classes and infrastructure to parse textual models into abstract syntax tree (AST) instances. These instances represent the content of models in a machine-processable way. For code generation, MontiCore provides a template-based code generation framework based on the FreeMarker template engine. This framework supports translating AST instances into arbitrary target representations, such as the Eclipse Modeling Framework, Mona, Java, or Python [23].

To facilitate modeling language engineering, MontiCore enables reusing language (parts) via inheritance, embedding, and aggregation [25]. *Inheritance* enables reusing productions from inherited grammars. This facilitates extending or specializing languages while reusing tooling (parsers, abstract syntax classes, context conditions, code generators) existing for the inherited language. With *embedding*, dedicated extension points in the host grammar are implemented by productions from embedded grammars. This enables reusing language (parts) for well-defined concerns, such as action languages or expression languages, in other languages. *Aggregation* loosely couples languages for joint analysis of their models. To this end, elements used in models of one language that reference elements of models of another language (such as the names referencing data types in an architecture language) are interpreted specific to the integration. As integration is external to both languages, it does not require participating languages to be aware of the integration.

3.2 MontiArcAutomaton

MontiArcAutomaton [20] is a modeling infrastructure for software architectures with exchangeable component behavior DSLs. It comprises the textual MontiArcAutomaton C&C ADL [21], which enables modeling architectures as hierarchies of components, and has been applied in industrial projects [28] and academic robotics contexts [23]. Components are connected via unidirectional connectors between the components' stable interfaces of typed ports. The types of ports are defined in UML/P [29] class diagrams, which is a variant of UML [27] class diagrams optimize for model-driven engineering. MontiArcAutomaton models distinguish between component types and instances, supports component configuration, generic type parameters, and inner components. Inner components resemble anonymous classes in Java, *i.e.,* they are defined within a composed component can be instantiated only in this context. The model of component `Localization` of Figure 2 is depicted in Figure 3.

The component `Localization` has an outgoing port `p` of data type `Pose` (l. 2), three subcomponents (ll. 3-5), and two explicit connectors (ll. 6-7). Subcomponent declarations consist of a component type name and a subcomponent name. The latter can be omitted to reduce the developers' cognitive load. Similarly, ports of the same name are connected automatically, thus corresponding connectors can be omitted as well.

```
01 component Localization {
02   port out Pose p;
03   component PoseFusion; // The names of subcomponent
04   component LIDAR;      // instances are derived
05   component GPS;        // automatically
06   connect lidar.p -> poseFusion.lp;
07   connect gps.p -> poseFusion.gp;
08   // Connector poseFusion.p -> p is derived also
09 }
```
MAA

Fig. 3: Textual model of the composed component `Localization` with three subcomponents.

3.3 Model-to-Model Transformations with MontiArcAutomaton

In model-driven development, M2M transformations are used to evolve, refactor, and normalize models. They can be more concise and better comprehensible than M2T transformations. We use the MATrans domain-specific transformation language (DSTL) for MontiArcAutomaton [30] to transform architecture models into representations better processable by subsequent M2T transformation. MATrans enables describing MontiArcAutomaton transformations in a problem-oriented fashion, using established vocabulary [31], and without the accidental complexity [2] of general transformation languages.

Figure 4 sketches the most important modeling elements and properties of MATrans: Names beginning with "$" are schema variables (l. 1), "$_" is an anonymous schema variable (l. 7), and the replacement operator "`:-`" (l. 11) replaces the pattern on its left by the pattern on its right and is delimited by double square brackets. Omitting the pattern on its left or on its right entails unconditional adding or removing, respectively. MATrans also supports specification of negative application conditions in form of negative elements, *i.e.,* elements that are forbidden in the model. Negative elements start with "`not`", followed by a model element enclosed in double square brackets (l. 10). While this example shows components at the top-level of the pattern, this is no prerequisite: all MontiArcAutomaton model elements can be pattern top-level elements, which eases specification of transformations.

```
01 component $source {
02   port out $type $name;
03 }
04 component $target {
05   port in $type $name;
06 }
07 component $_ {
08   component $source $subS;
09   component $target $subT;
10   not [[ connect $subS.$name -> $subT.$name; ]]
11   [[ :- connect $subS.$name -> $subT.$name; ]]
12 }
```
MATR

Fig. 4: Excerpt of a transformation that automatically connects ports of the same name in MontiArcAutomaton components.

Figure 4 matches three distinct components (ll. 1-12), two of which have compatible ports (ll. 2,5), and a composed component containing subcomponents of the former (ll. 7-12). The composed component must not contain a connector between the compatible ports of these subcomponents (ll. 8-9) as this transformation introduces it (l. 10).

3.4 The Robot Operating System (ROS)

ROS [6] is an infrastructure and framework for the efficient development of robotics applications. It comprises development tools and a messaging framework. Running ROS applications are flat graphs of GPL nodes and topics. Nodes are processes that perform computations and exchange the results via topics, which resemble typed message buses. The data types of topics are defined by `rosmsg`[2] models, which resemble a very restricted variant of class diagrams. Nodes publish and subscribe to topics in an event-driven fashion and may use libraries, frameworks, and APIs to compute behavior. Nodes have no types and can be reused as instances only. Topics are not defined explicitly, but by the publishers sending messages or the subscribers registering to these, *i.e.,* whether a topic exists is subject to the GPL code inside a node. Thus, without in-depth knowledge of nodes and their publishers, developing nodes that expect to receive messages from a specific topic is impossible. This hinders black-box reuse of nodes.

With ROS being a framework, the classes representing nodes must be implemented conforming to one of the ROS client library implementations in C++, Python, Lisp, or Java. Hence, software development with ROS nodes is subject to the "accidental complexities" [2] and "notational noise" [32] that arise from solving domain challenges with GPLs. Part of these accidental complexities arises from uncontrolled communication between nodes, which dynamically instantiate publishers and subscribers to interact with other nodes. However, what a node can receive and process is not declared in its interface, but part of its implementation only. Hence, node developers cannot rely on interfaces to compose nodes, but must investigate the source code of their implementations. Using an ADL with components of stable interfaces [8] to describe ROS graphs can facilitate this, but requires handling various idiosyncrasies of ROS including handling run-time connector reconfiguration as well as lacking generic data types in `rosmsg` and hierarchical nodes.

4 C&C Model Transformations

Model-to-model transformations [33] can facilitate architecture modeling by adjusting architectures to specific requirements to (1) facilitate subsequent processing steps; (2) reduce the cognitive load imposed on the modelers; and (3) instrument architectures for further analyses. In the following, we present M2M transformations identified useful for modeling robotics software architecture with MontiArcAutomaton as well as to facilitate code generation process from MontiArcAutomaton to ROS. New transformations for MontiArcAutomaton can be created and added easily as described in [30], [31]. As

2. http://wiki.ros.org/rosmsg

the adjustment is defined as a sequence of transformation rules (applied once or several times) the normalization is modular and can easily be extended or modified by removing transformation rules or adding new ones.

4.1 Eliminating Hierarchies

ROS describes software architectures as flat graphs of nodes and topics, whereas MontiArcAutomaton and many other C&C ADLs [11], [12] support describing hierarchical software architectures. While sophisticated code generators can translate hierarchical architectures into flat ROS artifacts, analysis of errors resulting from such transformation in the resulting GPL artifacts is subject to accidental complexities [2] and notional noise [32] again. Proper pre-processing can support analysis by flattening the architecture prior to code generation, hence enabling analysis on more abstract architecture model level instead.

Flattening architectures requires eliminating composed components. In MontiArcAutomaton and many other C&C ADLs, such components contain subcomponents and connectors, hence we focus on these elements. We use a transformation to successively disconnect composed components and reconnect their ports accordingly (*i.e.,*'lift' their connections). A subsequent transformation eliminates all unconnected components.

```
MATR
01 component $_ {
02   not [[ port $_ $_ ]]
03   component $interType $inter;
04   connect [[$inter.$iPort :- $atom.$aPort]] -> $_;
05   [[ :- component $atomType $atom; ]]
06 }
07
08 component $interType {
09   port out $portType $iPort;
10   component $atomType $atom;
11   connect $atom.$aPort-> $iPort;
12 }
13
14 component $atomType {
15   port out $portType $aPort;
16 }
17
18 assign {
19   $atom = uniqueName($atomType);
20 }
```

Fig. 5: A transformation to disconnect intermediate components prior to their elimination by a subsequent transformation.

The transformation first replaces connectors from subcomponents through intermediate components to their specific targets with a single connector from the subcomponent to its targets directly. Second, it eliminates the resulting empty hulls. The transformation depicted in Figure 5 takes care of the former. It considers three component types: the top-most component of the system architecture (ll. 1-6), the type of the intermediate subcomponent instance `$inter` to eliminate (l. 3), and the type of its atomic subcomponent `$atom`. Each connector from `$atom` to the interface of `$inter` to something on the environment of `$inter` is hence replaced by a connector from `$atom` to its target directly.

The transformation matches component types without ports (l. 2), the types of their intermediate subcomponents (l. 3), and related connectors (l. 4). The type of the intermediate subcomponent (ll. 8-12) yields an outgoing port (l. 9), contains a subcomponent of the atomic type (l. 10), and connects that subcomponent's port to its own outgoing port (l. 11). Afterwards, only the top-most architecture and atomic subcomponents exist. Please note that the complexity of this transformation is not due the transformation language, but the task at hand. Performing this transformation manually for a multitude of subcomponents is tedious and error prone. Re-implementing that for every code generator is costly as well.

4.2 Wrapping Port Data Types

Static C&C architectures, such as MontiArcAutomaton, fix the configuration of connectors at design time. While reducing flexibility, this establishes reliable communication in the sense that components cannot send and receive messages other than intended, which ultimately reduces development complexity. At the same time, middlewares such as ROS enable reconfiguring a system's architecture at runtime by flexibly rewiring connections and instantiating as well as deactivating nodes. To cope with this differences, the messages send between components are enveloped and the sender information is attached to the message. The generated ROS nodes accept messages from subscribed topics only, if the messages sender matches what was modeled in the architecture. The corresponding transformation depicted in Figure 6 takes care of this by replacing the type of each incoming port (l. 1) that is not yet wrapped by the wrapper type defined in the `assign` block (ll. 3-5). A similar transformation is applied to outgoing ports.

```
MATR
01 port in [[ $type :- $wrapper<$type> ]] $_;
02
03 assign {
04   $wrapper = "Envelope";
05 }
06
07 where {
08  $type != $wrapper
09 }
```

Fig. 6: Wrapping port types

This wrapping employs the data type `Envelope`, which yields a generic type parameter for the type of the message's payload, and instantiates it with the wrapped port's original data types. While helpful to add message meta-information easily, many middlewares, including ROS, do not support such generic type parameters.

4.3 Eliminating Generic Types

The type system of MontiArcAutomaton supports generic type parameters for component types and data types. This allows for greater flexibility than ROS. While ROS-specific refinements could be part of the code generation, encapsulating these into

a single M2M transformation (a) yields better comprehensible artifacts and (b) enables its reuse with multiple code generators. For instance, generic data types for ports and component configuration parameters in MontiArcAutomaton (similar to generics in Java or templates in C++) improve flexibility, however, subsequent translation into `rosmsg` types requires their replacement with specific types. Transformations can prepare architectures properly and provide developers a better overview on the resulting architecture than inspecting the produced ROS artifacts. The corresponding transformation replaces subcomponents whose component types rely on generic type parameters. As with generics in Java, subcomponents are parametrized with the actual types to be used at instantiation time. Hence in the actual software architecture, all generic type parameters have been assigned specific type arguments. To eliminate component types using generic type parameters from the architecture, the component types of such instances are be replaced by references to synthetic inner component types, where the generic types have been removed and replaced by the types assigned during instantiation.

```
01 component $name<$_> ComponentBody $BODY                          MATR
02
03 component $_ {
04   component [[ $name<$kind> :- $cName ]] $_;
05   [[ :- component $cName ComponentBody $PLAIN_BODY ]]
06   not [[ component $cName ComponentBody $PLAIN_BODY ]]
07 }
08
09 assign {
10   $cName = $name + "Of" + $kind;
11   $PLAIN_BODY = replaceGenerics($BODY, $kind);
12 }
```

Fig. 7: Replacing the types of subcomponents yielding generics by new component types with the generics eliminated.

Consequently, the transformation depicted in Figure 7 matches component types with generic type parameters indicated by angle brackets (l. 1) that are used in composed components (ll. 3-7) and replaces their types. It replaces their component types, which are parametrized by generic arguments (l. 4), with new inner component types (ll. 5-6). To this effect, it calculates a new component type name (l. 10) and a new component body (ll. 5-6), where occurrences of generic type arguments are replaced by the types the component was instantiated with (l. 4).

4.4 Automatically Connecting Ports

MontiArcAutomaton provides means to automatically connect ports under specific conditions (such as implicitly connected event ports in AADL [9]). Connectors are simple (*i.e.,* they do not have constraints or semantics aside from message passing) and connecting ports of adjacent subcomponent can be completed automatically if the ports have the same type. To this effect, MontiArcAutomaton provides two key phrases: 1) `autoconnect port` automatically connects the ports of subcomponent instances of a composed component based on their names and types. 2) `autoconnect type` similarly connects ports of subcomponent instances of a composed component based on their types only. The behavior of `autoconnect port` is illustrated in Figure 2. Here, `CleaningRobot` contains two subcomponents `Controller` and `Navigation` with matching ports that are not connected initially. This transformation connects these ports by an explicit connector (*cf.* bottom part of Figure 2). Figure 8 shows the transformation for the `autoconnect port` statement.

```
01 component $source {                                              MATR
02  port out $type $sName;
03 }
04
05 component $target {
06   port in $type $tName ]];
07 }
08
09 not [[ component $ambiguous {
10   port in $type $_;
11 } ]]
12
13 component $_ {
14   autoconnect type;
15   component $source $subS;
16   component $target $subT;
17   not [[ component $ambiguous $_; ]]
18   [[ :- connect $subS.$sName -> $subT.$tName; ]]
19 }
```

Fig. 8: Excerpt of a transformation that automatically connects ports of the same type in MontiArcAutomaton components.

In addition, MontiArcAutomaton provides shortcuts for connectors between ports of subcomponents. Instead of defining each connector between two subcomponents' ports individually, it is possible to define a connector between those subcomponents. In this case, all pairs of compatible connectors (i.e., ports of identical type) are connected automatically. The transformation rule is depicted in Figure 9. It matches two subcomponent instances (ll. 10-11), their related component definitions containing the ports (ll. 1-7), and a connector between the subcomponent instances (l. 12). Then it introduces connectors between ports of the same type if not already present. After this, a subsequent transformation rule removes all subcomponent connectors.

```
01 component $sType {                                               MATR
02   port out $pType $sPort;
03 }
04
05 component $tType {
06   port in $pType $tPort;
07 }
08
09 component $_ {
10   component $sType $source;
11   component $tType $target;
12   connect $source -> $target;
13
14   not [[ connect $source.$sPort -> $target.$tPort; ]]
15   [[ :- connect $source.$sPort -> $target.$tPort; ]]
16 }
```

Fig. 9: Excerpt of a transformation that automatically connects ports of MontiArcAutomaton subcomponents.

4.5 Normalizing Simple Connectors

MontiArcAutomaton allows modelers to attach connectors directly to subcomponent instance definitions. A simple connector is defined in square brackets right after the instances' name and connects a port of the instance with the defined target, *e.g.,* `component lidar [p->poseFusion.p]`. However, this is a shortcut for defining a normal connector and can thus be normalized before code generation.

```
01 component $_ {
02   component $_ $instance [
03     [[  $source -> $comp.$port :- ]]
04   ];
05    [[ :- connect $instance.$source -> $comp.$port; ]]
06 }
```
MATR

Fig. 10: Transformation to replace simple connectors with normal connectors.

The transformation rule for this transformation is shown in Figure 10. If first matches the simple connector (l. 3). This allows matching and removing the whole connector inside the replacement operator. The connector source and target are matched as well in order to introduce an equivalent normal connector (l. 5).

4.6 Completing Names

MontiArcAutomaton supports syntactic sugar and shortcuts regarding names. These can be reduced to other base concepts prior to code generation to reduce the complexity of the generator. For instance, enabling omitting superfluous names reduces notational noise [32] and supports developers in focusing on the important challenges. This pattern applies to many typical modeling elements of ADLs such as subcomponents, interface elements, configuration parameters, or constraints. With MontiArcAutomaton, an architecture modeler may omit the names of subcomponent instances in case there is just one subcomponent instance of its type in the containing component and ports if there is just one port of the corresponding type in the same component.

For instance, `component PoseFusion;` (l. 3 of Figure 3) will be assigned the derived instance name `poseFusion`, which can be used in the model without being made explicit, *e.g.,* for connectors (l. 6). Thus, the first two transformations add explicit default names, *i.e.,* the uncapitalized name of the subcomponent's or port's type, for each subcomponent instance or port, if not present. Both use the auxiliary method `uncapitalize()` to derive names in their `assign` blocks. Figure 11 depicts a realization of the transformation to add subcomponent instance names using MATrans. The transformation rule matches a subcomponent (ll. 1-3) that has no explicit name. This is ensured by the application constraint specified in the where block (ll. 7-9). In that case, a name for this subcomponent is added. The name is derived from the components type by uncapitalizing the first character. This is calculated in the assignment block (ll. 4-6).

```
01 $SC [[
02   component $type [[ :- $name ]];
03 ]]
04 assign {
05   $name = uncapitalize($type);
06 }
07 where  {
08   $SC.getInstanceNames().isEmpty()
09 }
```
MATR

Fig. 11: Deriving default names for subcomponent instances.

Figure 12 depicts the transformation for incoming ports. First, an incoming port is matched (l. 1). The application constraint expressed in the where block ensures that this port has no explicit name yet (ll. 5-7). In that case, a name for this port is added, which is derived from the ports type by uncapitalizing the first character. Again, this is calculated in the assignment block (ll. 2-4).

```
01 port $PL [[  in $type [[ :- $name ]] ];
02 assign {
03   $name = uncapitalize($type);
04 }
05 where {
06   $PL.getName().isEmpty()
07 }
```
MATR

Fig. 12: Deriving default names for incoming ports.

With MATrans, inner MontiArcAutomaton components (similar to anonymous classes in Java) can be easily instantiated automatically: if no instance of the corresponding type exists, such an instance is added to the comprising parent component. Figure 13 depicts this transformation. MontiArcAutomaton furthermore allows naming inner components explicitly which is a shortcut for defining an inner component and declaring an instance of it. A further transformation normalizes this by adding an explicit instance of the named inner component with the specified name and removes the name from the inner component.

```
01 component $_ {
02   component $typeName { /* ... */ }
03   not [[ component $typeName $_; ]]
04   [[ :- component $typeName $instanceName; ]]
05 }
06 assign {
07   $instanceName = uncapitalize($typeName);
08 }
```
MATR

Fig. 13: Instantiation of inner components.

4.7 Adding Run-time Inspection Infrastructure

We also support automated integration of run-time inspection infrastructure via M2M transformations. The corresponding transformations 1) adds a monitor subcomponent to every composed component, 2) adds outgoing ports to all subcomponents of each composed components, which emit messages describing the subcomponents' states, and connects their instances to the monitoring component. 3) This enables moni-

toring subcomponents as described in [30] without integrating all these components, ports, and connectors manually.

To this end, this transformation iterates over all components of a software system and applies three transformation rules. The first transformation rule is shown in Figure 14. It adds a state ports to all components (l. 2) if not present yet (l. 3). The name of this port is composed of the name of the component and the suffix `State` (ll. 5-7).

```
01 component $name {
02   port [[ :- out $sp state ]],
03   not [[ out $_ state ]];
04 }
05 assign {
06   $sp = $name.concat("State");
07 }
```
MATR

Fig. 14: Adding state ports to components.

The second transformation rule is depicted in Figure 15. It adds a monitoring component (l. 2) to components that have subcomponents (l. 4) if not already present (l. 3). The type of the monitor is composed of the name of the component whose subcomponents are monitored and the suffix `Monitor` (ll. 6-8).

```
01 component $name {
02   [[ :- component $type monitor {}]]
03   not [[ component $_ monitor {} ]]
04   component $_ $sub {}
05 }
06 assign {
07   $type = $name.concat("Monitor");
08 }
```
MATR

Fig. 15: Adding a monitoring component.

Finally the third transformation rule (depicted in Figure 16) connects the monitoring component to the state ports introduced in the upstream transformation rules. To this end, for every subcomponent (l. 2) the transformation rule adds an incoming port to the monitoring component (ll. 3-5). The name and type of the port is calculated in the assign-block (ll. 8-11). Furthermore, a connector connecting the state port of the subcomponent with the newly introduced state port of the monitoring component is added (l. 6).

```
01 component $_ {
02   component $_ $name;
03   component $_ monitor {
04     port [[ :- in $type $sp]];
05   }
06   [[ :- connect $name.state -> monitor.$sp; ]]
07 }
08 assign {
09   $type = $name.concat("State");
10   $sp = uncapitalize($type);
11 }
```
MATR

Fig. 16: Connecting the monitoring component.

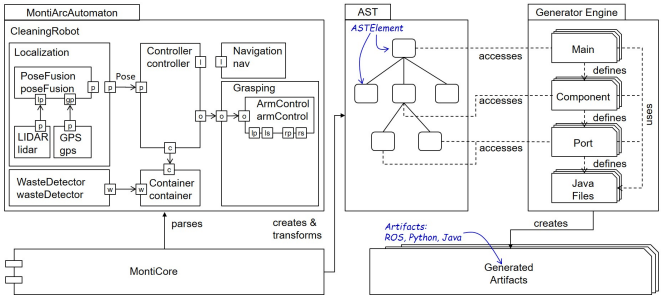

Fig. 17: Quintessential modules and artifacts of the generator engine with its provided internal model representation in form of an AST.

5 Transforming Component & Connector Architectures to ROS

Model-to-Text transformations foster model-driven development by aligning platform-agnostic software architectures with executable software systems. In many complex domains, such as robotics, distributed system architectures must be 1) tailored to include expertise encoded in middlewares, 2) implemented in supported GPLs, and 3) adjusted to specific runtime environments. Bridging the gap between logical design concepts and middleware-specific implementation concepts is challenging for M2T transformations. In the following, we investigate challenges for M2T transformations, elaborate on expenditure reduction for code generator development, and provide as proof-of-concept a M2T transformation from C&C architectures to the Python implementation of ROS Groovy.

5.1 Challenges for M2T Transformations

Architecture models conform to particular ADLs. Each ADL is designed for a specific purpose supporting specific features. Nonetheless, there is some consensus [11] in common ADL modeling elements [11] that originates from their common background of component-based software engineering [7]. This includes providing modeling elements for components, connectors, and configurations. Preserving the semantics and properties of the modeling elements in M2T transformations is challenging. Especially, when the ADL design concepts differ significantly from the concepts of the targeted middleware or GPL, the required M2T transformations often become very complex.

Code generators implement such M2T transformations and represent a systematic approach for automated M2T transformations. However, code generators are tailored to a specific middleware and usually are not reusable for different target platforms. With multiple target platforms, each transformation step has to be developed for each code generator (which could even employ different implementation technologies). For instance, the MontiArcAutomaton ADL supports hierarchical components, whereas its Java implementation for embedded systems as well as ROS support flat graphs only.

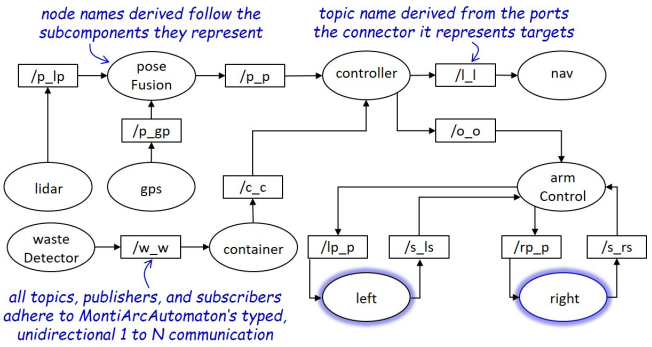

Fig. 18: ROS graph of the cleaning robot architecture depicted in Figure 2.

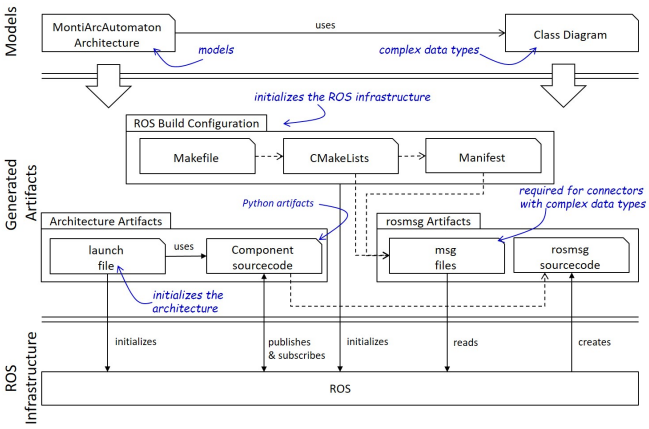

Fig. 19: Overview of the generated infrastructure.

Implementing generators for both target platforms requires defining structure flattening transformation steps for each code generator separately. To reduce implementation effort, it is beneficial to lift the flattening operation to the model level beforehand.

However, such pre-processing M2M transformations must ensure that the output model conforms to the underlying ADL again.

5.2 Generator Engine for M2T Transformations

We employ the FreeMarker template engine and the MontiArcAutomaton code generation framework [20] to transform MontiArcAutomaton architecture models to ROS nodes, topics, and configuration artifacts.

Figure 17 illustrates an overview of the generation process. Starting with an MontiArcAutomaton architecture model for a cleaning robot, the MontiCore language workbench parses the model into abstract syntax tree (AST) instances, which represent the content of the models in a machine-processable form. In MontiArcAutomaton, each element of the AST represents a specific architectural element. While M2M transformations are applied on the AST beforehand, the `Generator Engine` accesses the transformed AST and its elements. This separation of M2M and M2T transformations foster engineers to develop multiple generator engines (each for a different middleware) on basis of a common model representation. For instance, we employed a generator engine for Java and ROS Python on the provided AST. Both generator engines consists of three kinds of elements: a dedicated main template, subtemplates (*e.g.,* `Component` and `Port`), and Java artifacts. The main template defines the required subtemplates, Java artifacts, as well as additional properties required for the generation process. To this effect, the AST is accessible to the generator developer in the main template. Elements of the AST might be passed to relevant subtemplates and Java artifact calls. For instance, the AST subtree of an atomic MontiArcAutomaton component is passed to the `Component` subtemplate. The Java artifacts operate on different elements of the AST. Intended side-effects, such as inferred values that occurred within the template or Java artifact call, are stored in a Java artifact and accessible for the generator developer within the templates. Whereas templates defined in the generator engine are specific to middleware and target language, the Java artifacts are specific to the AST and can be reused by different generator engines.

5.3 Transforming C&C Architectures to ROS

The prime concerns of transforming C&C architectures to ROS are 1) translating the atomic and composed component types to ROS nodes with publishers and subscribers; 2) translating connectors to ROS topics; 3) translating the class diagram data types that are used for ports to `rosmsg` models; and 4) creating the configuration files required for ROS projects. The intended result of transforming the `CleaningRobot` software architecture (*cf.* Figure 2) with our generator engine is presented in Figure 18. Components and connectors are translated to nodes and topics. Node names are derived follow the subcomponent instances they present. Topic names are combinations of the connector names and the participating source and target ports. Each topic, publisher, and subscriber adhere to MontiArcAutomaton's typed, unidirectional communication. For instance, the subcomponent `poseFusion` of type `PoseFusion` is translated to a Python class artifact that defines a ROS node and a single publisher to the topic `p_p`, whose name and type are derived from the connector between the `poseFusion` and `controller` component instances.

Deploying the architecture on ROS and simultaneously preserve the presented ROS graph requires to generate various artifacts as depicted in Figure 19: From the MontiArcAutomaton architecture and class diagram models, we produce the necessary configuration artifacts (`Makefile`, `CMakeLists`, `Manifest`) to initialize the ROS building process, the artifacts representing the architecture in ROS Python, and the `rosmsg` artifacts. The following sections elaborate on the generated artifacts and their relation to ROS.

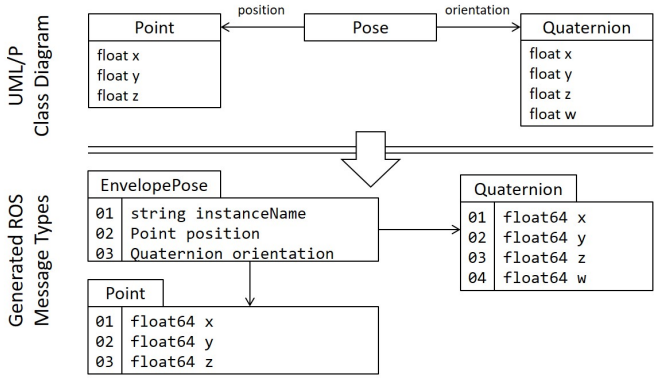

Fig. 20: Example of ROS message artifacts derived from types defined in UML/P class diagrams.

5.3.1 ROS Messages

MontiArcAutomaton architectures operate in the context of UML/P [29] class diagrams, hence these must be translated as well. MontiArcAutomaton supports the full expressiveness of UML/P class diagrams, *e.g.,* interfaces, abstract classes, and generic type parameters. ROS nodes operate in the context of simpler `rosmsg` models, which do not support these. Consequently, class diagram port types using these features are prohibited for translation to ROS and our code generator takes care of rejecting such models. Aside from these challenges, the classes are as follows: primitive attributes becomes `rosmsg` properties, attributes of complex types become nested properties, 1-to-n relations become arrays of variable length.

Figure 20 illustrates exemplary derived ROS message artifacts from complex types defined in an UML/P class diagram. The depicted `Pose` data type is, for instance, used between the `poseFusion` and the `controller` component instances (*cf.* Figure 2). To this effect, this type is transformed during the M2M transformation to the `Envelope<Pose>` data type and, subsequently, translated into the `EnvelopePose.msg` artifact, which is a ROS specific message type. While the `Pose` data type is wrapped in an `Envelope` type, the generator does not wrap the associated types (*e.g.,* `Quaternion` and `Point`) into `Envelope` types but translates them to ROS specific messages types. Each containing primitive ROS messages types. Ultimately, the ROS build system generates client specific artifacts that can be used in the generated component type artifacts.

5.3.2 ROS Nodes

Transforming MontiArcAutomaton models to ROS nodes is more challenging. Its various concepts, such as component types, instances, parameters, ports, and connectors, must be translated to concepts available to ROS Python. For instance, MontiArcAutomaton realizes component-based software engineering by enforcing that components are black-boxes unaware of their environment aside from messages passed to their inputs. Consequently, its component instances are unaware of their communication partners. Instead, the containing components define connectors between their subcomponents. In ROS, nodes exist in flat graphs, *i.e.,* there is no containing component, and they are aware of their environment in terms of topics that can be subscribed and published to – which hampers reuse. Hence, the transformation of MontiArcAutomaton components into ROS nodes must integrate this information into their implementations. However, as MontiArcAutomaton components may be used in different contexts – and hence with different communication partners – encoding the subscribed and published topics into the generated Python artifacts is not feasible.

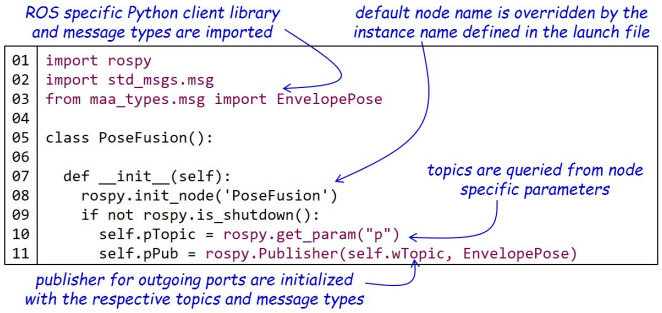

Fig. 21: Excerpt of a generic `PoseFusion` component type implementation. Violet marked code represent ROS specific relations.

Instead, each MontiArcAutomaton component type (*e.g.,* the `PoseFusion` component type in Figure 21) becomes a Python class with generic publish and subscribe mechanisms (ll. 10,11), which initializes a single node (l. 8), defines a single publisher for each outgoing port (l. 11), a single subscriber for each incoming port, and defines parameters for each component configuration parameter. Node name, topic, and parameters can be defined at constructing instances of this class and hence allow to reuse it similar to MontiArcAutomaton components in different communication contexts. To this effect, the node name defined in class (l. 8) is a default name that will be overridden by the instance name defined in a launch file. ROS specific client libraries and message types, as well as generated message type, are provided for ports and topics (ll. 1-3). These classes also implement rejecting enveloped messages received from senders other than configured in the architecture model. With this in place, connectors are translated into topics, such that each topic realizes the connection of exactly one source port to one target port of the architecture model.

The information on instance specific node names, connected topics, and available parameters is generated into roslaunch[3] configuration files. These are also generated from the MontiArcAutomaton architecture and take care of instantiating the

3. http://wiki.ros.org/roslaunch/XML

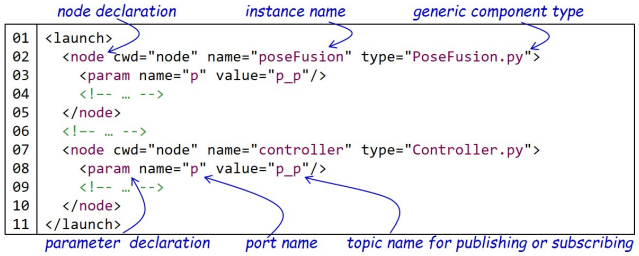

Fig. 22: Excerpt of the derived `roslaunch` file from the architecture depicted in Figure 2. Violet marked code represent ROS specific declarations.

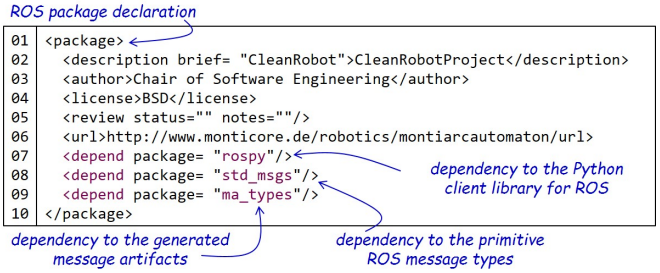

Fig. 23: Generated `Manifest.xml` artifact. Violet marked code represent ROS specific dependencies.

generated Python classes according to the architecture model, *i.e.,* they name, connect and parametrize the node instances as governed by their related component instances. For instance, Figure 22 illustrates an excerpt of the derived `roslaunch` artifact from the architecture depicted in Figure 2. Each node declaration starts with a `node` tag containing parameters, such as name and type (ll. 2,7). Both are derived from the associated component instance. As the generated component type artifacts are generic to the subscribed and published topics, the node declaration encloses node specific parameters composing a name and a value. For instance, the component instance `poseFusion` publishes via its port `p` to topic `p_p` (l. 3). Contrary to the `poseFusion` instance, the `controller` component instance subscribes to the same topic. Both component types infer their node parameters at instantiation time via ROS and are able to allocate the parameter values for their respective incoming and outgoing ports. Parameters for component instance are defined analogously. To ensure that there are no duplicated parameters, port and parameter names must be unique, which is ensured by previous M2M transformations.

5.3.3 ROS Build Configuration

The overall concept of ROS is to develop robotics applications by executing and sharing coherent code units. These units are grouped in packages, integrated in ROS, and accessible for further packages in form of dependencies. To this effect, the integrated build system of ROS (named `rosbuild`) enables developers to establish new ROS packages and to integrate them into ROS by means of three configuration files: `Manifest.xml`, `CMakeLists.txt`, and `MakeFile`. Each of is produced by the generator engine and the information derived from the projects architecture.

For instance, the `Manifest.xml` in Figure 23 declares the ROS package for the architecture including the required package information and the dependencies to other ROS packages. The ROS package declaration within the `Manifest.xml` artifact contains package specific information (ll. 2-6), such as a brief description, the author of the packages, the license, the review status, and the URL for further package descriptions. While these elements are required in the `Manifest.xml` artifact, the subsequent package dependency definitions (ll. 7-9) are in general optional in the `Manifest.xml` but required in our package declaration. The generator engine extracts the information about the target language from the `Main` template and integrates a dependency to the target language specific ROS library client, for instance, the Python client library `rospy` (l. 7). This library enables to orchestrate ROS from within Python artifacts (*e.g.,* establishing nodes, subscribing and publishing to topics, and receiving/passing parameters from/to nodes). In addition, the generator engine adds dependencies to the packages for primitive ROS message types (l. 8) and more complex data types (l. 9) derived from the UML/P class diagram.

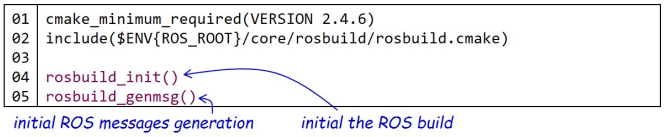

Fig. 24: Generated `CMakeLists` artifact. Violet marked code represent ROS specific macros.

As the ROS build system uses CMake[4] to build its packages, a CMake specific configuration (`CmakeLists.txt`) is generated for each architecture. Figure 24 illustrates the generated artifact that must contain the information about the required minimum version of CMake (l. 1) and an `include` statement (l. 2), which enables to use ROS build specific macros, such as `rosbuild_init()` (l. 4) and `rosbuild_genmsg` (l. 5). The first macro is used to configure default input/output directories and the compilation process. The latter macro processes our generated ROS message definitions by means of a client library specific code generator to make them accessible, for instance, to the nodes and topics.

Ultimately, an engineer that wants to invoke the ROS build process must use the `rosmake` command in a ROS supporting terminal. As this process still uses make to build the packages, a `MakeFile` has to be provided that invokes CMake.

4. https://cmake.org/

5.4 Evaluation of the M2T Transformations

The resulting code generator enables translating MontiArcAutomaton architecture models to ROS Python nodes. Combining the benefits of integrating existing solutions encoded in middleware artifacts with the benefits of architecture modeling therefore becomes straightforward: Architectures can feature components with unconnected ports. Via translation to topics, the publishers and subscribers resulting from transformation can easily interact with middleware artifacts for which no component models exist (for instance, via configuration in roslaunch files). Thus the encoded expertise can be reused without giving the employed ADLs benefits (such as up stable interfaces or hierarchical component topologies). To evaluate our ROS Python generator, we implemented a similar code generator to translate MontiArcAutomaton components into plain Java artifacts [23]. The Java generator encodes all transformation steps in Java and FreeMarker and consequently the ROS Python generator is significantly less complex. As illustrated in Table 1, the Java generator comprises more than twice as many FreeMarker templates and Java classes. However, the templates of both code generators are, in average, of the same lengths and the ROS Python generator's Java classes are only a little larger.

TABLE 1: Artifacts of two similar generators: translation to ROS Python uses M2M transformations, the other does not.

Generator	# Templates	Avg. LOC	# Classes	Avg. LOC
ROS Python	18	39.6	14	106.8
Plain Java	40	40.7	24	90.6

All of this is enabled by six M2M transformations formulated in a language that closely resemble the MontiArcAutomaton ADL. We thus believe that decoupling code generation from ADL development and usage via appropriate model transformations can greatly facilitate development of robotics modeling tool chains and, ultimately, robotics software.

6 Discussion and Related Work

Applying the presented method requires expertise in various challenging fields, including software language engineering, model transformation, and code generator development. It is, however, not primarily aimed at architecture modelers, but at tool chain providers developing architecture modeling solutions with code generation capabilities, such as SmartSoft [16] or DiaSpec [15]. In such contexts, expertise in language engineering and model transformation already exists. However, with the proposed separation of concerns, the challenge of providing a middleware-specific code generator can be separated into (1) creating less complex, yet middleware-specific M2T transformations and (2) providing proper, ADL-specific M2M transformations suitable for code generation. This enables reusing expertise encoded in existing middleware modules easily. Our approach differs from the OMG's model-driven architecture [34] in focusing on tool chain modularity: it does not prescribe that the transformed architecture models are more platform-specific.

Related architecture modeling infrastructures in robotics focus on domain challenges over infrastructure modularity and reuse of middleware-compatible artifacts, such as ROS [6] nodes or Orocos [4] components. For instance, the DiaSpec [15] infrastructure comprises an ADL with different component kinds, but does neither support exchangeable model transformations nor exchangeable code generators. The SmartSoft [16] infrastructure also comprises an ADL, integrated model transformations, means for behavior modeling, contingency planning and – based on Xtext – generally enables integration of further code generation capabilities. However, it also does not support exchanging its M2M transformations and integration of further code generation is not investigated yet. The authors of [35] propose modeling self-adaptive software with components and translating it to Fractal [36] component implementations that neither supports extensible M2M, nor exchanging the code generators. RobotML [37] is a UML profile for modeling structure, behavior, and communication of robot software architectures implemented with as a UML profile for the Papyrus[5] modeling environment. It uses Acceleo[6] for code generation and, thus, should in principle support exchangeable code generators as well. It, however, does not support extensible M2M transformations.

7 Conclusion

We have presented a model-driven method for separating the concerns of architecture modelers from the technical concerns of code generator developers. This method leverage modular domain-specific M2M transformations and relies on gradually transforming the architecture under development into representations better processable by subsequent transformation tools such as code generators. This greatly reduces the effort of developing code generators to interface with specific middlewares and ultimately facilitates model-driven development of robotics architectures by enabling to reuse expertise encoded in middleware artifacts, such as ROS nodes.

We applied this method to the transformation of MontiArcAutomaton components into nodes of the Python implementation of ROS Groovy. To this effect, we presented the employed reusable M2M transformations and showed how they facilitate generator development by comparison to a generator producing Java implementations without using M2M transformations. The resulting ROS Python generator is significantly less complex than the Java generator, which indicates that separating concerns by employing M2M transformations is beneficial in creating middleware-specific code generators. We

5. https://eclipse.org/papyrus/
6. http://www.eclipse.org/acceleo/

believe, this separation can produce better extensible model-driven tool chains in robotics and, hence, ultimately facilitate model-driven development in robotics.

REFERENCES

[1] A. Nordmann, N. Hochgeschwender, and S. Wrede, "A Survey on Domain-Specific Languages in Robotics," in *International Conference on Simulation, Modeling, and Programming for Autonomous Robots*, Bergamo, 2014. 1

[2] R. France and B. Rumpe, "Model-Driven Development of Complex Software: A Research Roadmap," in *Future of Software Engineering 2007 at ICSE.*, 2007. 1, 3.3, 3.4, 4.1

[3] M. Völter, T. Stahl, J. Bettin, A. Haase, S. Helsen, and K. Czarnecki, *Model-Driven Software Development: Technology, Engineering, Management*, ser. Wiley Software Patterns Series. Wiley, 2013. 1

[4] H. Bruyninckx, "Open Robot Control Software: the OROCOS project," in *2001 ICRA IEEE International Conference on Robotics and Automation (ICRA)*, vol. 3. IEEE, 2001, pp. 2523–2528. 1, 6

[5] I. A. Nesnas, A. Wright, M. Bajracharya, R. Simmons, T. Estlin, and W. S. Kim, "CLARAty: an architecture for reusable robotic software," pp. 253–264, 2003. 1

[6] M. Quigley, B. Gerkey, K. Conley, J. Faust, T. Foote, J. Leibs, E. Berger, R. Wheeler, and A. Ng, "ROS: an open-source Robot Operating System," in *ICRA Workshop on Open Source Software*, 2009. 1, 1, 3, 1, 3.4, 6

[7] P. Naur and B. Randell, Eds., *Software Engineering: Report of a conference sponsored by the NATO Science Committee, Garmisch, Germany, 7-11 Oct. 1968, Brussels, Scientific Affairs Division, NATO*, 1969. 1, 5.1

[8] D. Brugali and P. Salvaneschi, "Stable Aspects In Robot Software Development," *International Journal of Advanced Robotic Systems*, vol. 3, 2006. 1, 3.4

[9] P. H. Feiler and D. P. Gluch, *Model-Based Engineering with AADL: An Introduction to the SAE Architecture Analysis & Design Language*. Addison-Wesley, 2012. 1, 4.4

[10] V. Debruyne, F. Simonot-Lion, and Y. Trinquet, "An Architecture Description Language," in *Architecture Description Languages*. Springer, 2005, pp. 181–195. 1

[11] N. Medvidovic and R. Taylor, "A Classification and Comparison Framework for Software Architecture Description Languages," *IEEE Transactions on Software Engineering*, 2000. 1, 4.1, 5.1

[12] I. Malavolta, P. Lago, H. Muccini, P. Pelliccione, and A. Tang, "What Industry Needs from Architectural Languages: A Survey," *IEEE Transactions on Software Engineering*, 2013. 1, 4.1

[13] H. Bruyninckx, M. Klotzbücher, N. Hochgeschwender, G. Kraetzschmar, L. Gherardi, and D. Brugali, "The BRICS Component Model: A Model-Based Development Paradigm For Complex Robotics Software Systems," in *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, 2013. 1

[14] F. J. Ortiz, F. Sánchez, D. Alonso, F. Rosique, and C. C. Insaurralde, "C-Forge: a Model-Driven Toolchain for Developing Component-Based Robotics Software," in *Proceedings of IEEE ICRA 2013 - Workshop Software Development and Integration in Robotics (SDIR VIII)*, 2013. 1

[15] D. Cassou, P. Koch, and S. Stinckwich, "Using the DiaSpec design language and compiler to develop robotics systems," in *Proceedings of the Second International Workshop on Domain-Specific Languages and Models for Robotic Systems (DSLRob 2011)*, 2011. 1, 6, 6

[16] C. Schlegel, A. Steck, and A. Lotz, "Model-Driven Software Development in Robotics : Communication Patterns as Key for a Robotics Component Model," in *Introduction to Modern Robotics*. iConcept Press, 2011. 1, 6, 6

[17] D. Alonso, C. Vicente-Chicote, F. Ortiz, and J. Pastor, "V3CMM : a 3-View Component Meta-Model for Model-Driven Robotic Software Development," *Journal of Software Engineering for Robotics (JOSER)*, vol. 1, no. January, pp. 3–17, 2010. 1

[18] A. Haber, J. O. Ringert, and B. Rumpe, "Towards Architectural Programming of Embedded Systems," in *Tagungsband des Dagstuhl-Workshop MBEES: Modellbasierte Entwicklung eingebetteter Systeme VI*. Munich, Germany: fortiss GmbH, February 2010, pp. 13–22. 1

[19] A. Navarro Pérez and B. Rumpe, "Modeling Cloud Architectures as Interactive Systems," in *Proceedings of the 2nd International Workshop on Model-Driven Engineering for High Performance and Cloud Computing*, 2013. 1

[20] J. O. Ringert, A. Roth, B. Rumpe, and A. Wortmann, "Language and Code Generator Composition for Model-Driven Engineering of Robotics Component & Connector Systems," *Journal of Software Engineering for Robotics (JOSER)*, 2015. 1, 1, 2, 3.2, 5.2

[21] J. O. Ringert, B. Rumpe, and A. Wortmann, *Architecture and Behavior Modeling of Cyber-Physical Systems with MontiArcAutomaton*. Shaker Verlag, 2014. 1, 3.2

[22] K. Adam, K. Hölldobler, B. Rumpe, and A. Wortmann, "Engineering Robotics Software Architectures with Exchangeable Model Transformations," in *International Conference on Robotic Computing (IRC'17)*. IEEE, April 2017, pp. 172–179. 1

[23] J. O. Ringert, B. Rumpe, and A. Wortmann, "From Software Architecture Structure and Behavior Modeling to Implementations of Cyber-Physical Systems," in *Software Engineering 2013 Workshopband*, 2013. 2, 3.1, 3.2, 5.4

[24] H. Krahn, B. Rumpe, and S. Völkel, "Monticore: a framework for compositional development of domain specific languages," in *International Journal on Software Tools for Technology Transfer (STTT)*, 2010. 3.1

[25] A. Haber, M. Look, P. Mir Seyed Nazari, A. Navarro Perez, B. Rumpe, S. Völkel, and A. Wortmann, "Composition of Heterogeneous Modeling Languages," in *Model-Driven Engineering and Software Development Conference (MODELSWARD'15)*, ser. CCIS, vol. 580. Springer, 2015, pp. 45–66. [Online]. Available: http://www.se-rwth.de/publications/Composition-of-Heterogeneous-Modeling-Languages.pdf 3.1

[26] S. Erdweg, T. van der Storm, M. Vlter, M. Boersma, R. Bosman, W. R. Cook, A. Gerritsen, A. Hulshout, S. Kelly, A. Loh, G. D. Konat, P. J. Molina, M. Palatnik, R. Pohjonen, E. Schindler, K. Schindler, R. Solmi, V. A. Vergu, E. Visser, K. van der Vlist, G. H. Wachsmuth, and J. van der Woning, "The State of the Art in Language Workbenches," in *Software Language Engineering*. Springer International Publishing, 2013. 3.1

[27] Object Management Group, "OMG Unified Modeling Language (OMG UML), Superstructure Version 2.3 (10-05-05)," May 2010, http://www.omg.org/spec/UML/2.3/Superstructure/PDF/ [Online; accessed 2017-07-14]. 3.1, 3.2

[28] R. Heim, P. Mir Seyed Nazari, J. O. Ringert, B. Rumpe, and A. Wortmann, "Modeling Robot and World Interfaces for Reusable Tasks," in *2015 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2015)*, 2015. 3.2

[29] B. Rumpe, *Modeling with UML: Language, Concepts, Methods*. Springer International, July 2016. [Online]. Available: http://www.se-rwth.de/mbse/ 3.2, 5.3.1

[30] L. Hermerschmidt, K. Hölldobler, B. Rumpe, and A. Wortmann, "Generating Domain-Specific Transformation Languages for Component & Connector Architecture Descriptions," in *2nd International Workshop on Model-Driven Engineering for Component-Based Software Systems (ModComp) 2015*, 2015. 3.3, 4, 4.7

[31] K. Hölldobler, B. Rumpe, and I. Weisemöller, "Systematically Deriving Domain-Specific Transformation Languages," in *Conference on Model Driven Engineering Languages and Systems (MODELS'15)*, 2015. 3.3, 4

[32] D. S. Wile, "Supporting the DSL Spectrum," *Computing and Information Technology*, 2001. 3.4, 4.1, 4.6

[33] T. Mens, K. Czarnecki, and P. V. Gorp, "A Taxonomy of Model Transformations," in *Language Engineering for Model-Driven Software Development*, J. B. und Reiko Heckel, Ed. Dagstuhl Seminar Proceedings. Internationales Begegnungs- und Forschungszentrum (IBFI), Schloss Dagstuhl, Germany, 2005. 4

[34] Object Management Group, "MDA Guide Version 1.0.1," June 2003, http://www.omg.org/news/meetings/workshops/UML_2003_Manual/00-2_MDA_Guide_v1.0.1.pdf [Online; accessed 2015-12-17]. 6

[35] J. F. Inglés-Romero, C. Vicente-Chicote, B. Morin, and O. Barais, "Towards the Automatic Generation of Self-Adaptive Robotics Software: an Experience Report," in *Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE), 2011 20th IEEE International Workshops on*. IEEE, 2011, pp. 79–86. 6

[36] E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, and J.-B. Stefani, "The FRACTAL component model and its support in Java," *Software, Practice, and Experiance*, vol. 36, no. 11-12, pp. 1257–1284, 2006. 6

[37] S. Dhouib, S. Kchir, S. Stinckwich, T. Ziadi, and M. Ziane, "RobotML, a Domain-Specific Language to Design, Simulate and Deploy Robotic Applications," in *Simulation, Modeling, and Programming for Autonomous Robots*, ser. Lecture Notes in Computer Science, I. Noda, N. Ando, D. Brugali, and J. Kuffner, Eds. Springer Berlin Heidelberg, 2012, vol. 7628, pp. 149–160. 6

Kai Adam received his B. Sc. and M. Sc. degrees in computer science from the RWTH Aachen University, in 2013 and 2016. Currently, he is a research assistant and Ph.D. candidate at the Department of Software Engineering at RWTH Aachen University. His research interests covers software engineering, model-driven development, model-based testing, software architectures.

Katrin Hölldobler received her B. Sc. and M. Sc. degrees in computer science from RWTH Aachen University in 2010 and 2012. Currently, she is a research assistant and Ph.D. candidate at the Department of Software Engineering at RWTH Aachen University. Her research interest covers software engineering, design and development of domain-specific transformation languages, software language engineering, and model transformation. She is a member of ACM.

Bernhard Rumpe is chair of the Department for Software Engineering at the RWTH Aachen University, Germany. His main interests are software development methods and techniques that benefit from both rigorous and practical approaches. This includes the impact of new technologies such as model-engineering based on UML-like notations and domain-specific languages and evolutionary, test-based methods, software architecture as well as the methodical and technical implications of their use in industry. He has furthermore contributed to the communities of formal methods and UML. Since 2009 he started combining modeling techniques and cloud computing. He is author and editor of eight books and editor-in-chief of the Springer International Journal on Software and Systems Modeling. See http://www.se-rwth.de/topics/ for more.

Andreas Wortmann received his Ph.D. from RWTH Aachen University in 2016. Currently, he is a tenured researcher at the Department for Software Engineering at RWTH Aachen University. His research interests cover software engineering, software architectures, model-driven development, robotics, and software language engineering. He is a member of IEEE and its Technical Committee on Software Engineering for Robotics and Automation.