Engineering Robotics Software Architectures with Exchangeable Model Transformations

Kai Adam, Katrin Hölldobler, Bernhard Rumpe, and Andreas Wortmann
Software Engineering
RWTH Aachen University
www.se-rwth.de

***Abstract*—Robotics has adopted modeling with architecture description languages (ADLs). This introduces a gap when reusing solutions encoded in middleware modules. Existing ADL modeling in robotics focuses on domain challenges instead of tool modularity, hence customizing an ADL tool to generate solutions conforming to a specific middleware (e.g., ROS) is challenging. This could produce a multitude of incompatible 'vendor-locked' tool 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 using the extensible tool chain of MontiArcAutomaton, which enables translating software architecture models gradually into middleware modules using exchangeable model-to-model and model-to-text transformations. Employing architecture modeling with modular tool chains enables combining the benefits of ADLs with the solutions encoded in popular middlewares and ultimately facilitates robotics software engineering.**

I. Introduction

Robotics is one of the most challenging domains for software engineering. Successful deployment of even simple robotics applications requires expertise from multiple domains and combination of heterogeneous software solutions. Robotics has successfully adopted [1] model-driven development (MDD) [2], which facilitates the integration of domain experts by lifting abstract and better comprehensible models to primary development artifacts. There is, however, a big corpus of robotics solutions encoded in general programming language (GPL) artifacts that are tailored to specific middlewares (such as Orocos [3], CLARAty [4], or ROS [5]). These are hardly accessible by MDD solutions. On the other hand, the reuse promised by component-based software engineering has been identified crucial to reusable robotics architectures [6]. Similar to avionics [7] and automotive [8], robotics-specific architecture description languages (ADLs) [9], [10] lift the notion of components to *component models* (cf., BRICS [11], C-Forge [12], DiaSpec [13], SmartSoft [14], or V3CMM [15]). 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 modeling software architectures for automotive [16], cloud systems [17], and robotics [18], we present an extensible architecture modeling method that employs exchangeable model-to-model (M2M) and model-to-text (M2T) transformations to enable translating architecture models into implementations for arbitrary target middlewares. To this effect, it separates the concerns of architecture modelers from the concerns of model transformation developers and code generator engineers as depicted in Fig. 1.

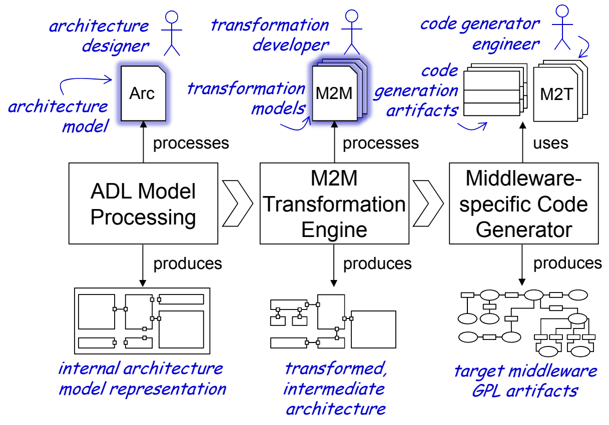
architecture designer
transformation developer
code generator engineer
architecture model
Arc
transformation models
M2M
code generation artifacts
M2T
processes
processes
uses
ADL Model Processing
M2M Transformation Engine
Middleware-specific Code Generator
produces
produces
produces
internal architecture model representation
transformed, intermediate architecture
target middleware GPL artifacts

Fig. 1. Quintessential components, artifacts, and roles for pervasive, modular architecture modeling for different target middlewares illustrated on example of the translation to ROS [5].

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 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 [18], [19], 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 [5]

In the following, Sect. II motivates the benefits of modular architecture modeling by example, before Sect. III presents preliminaries. Afterwards, Sect. IV describes the modular M2M transformations and Sect. V describes M2T translation to ROS. Sect. VI discusses observations and related work. Sect. VII concludes.

II. Example

Consider a company producing the software architecture for a cleaning robot with two arms as depicted in Fig. 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 [5] for execution; and (2) to Java for simulation [20]. 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 [18]. The resulting ROS nodes can easily interface with existing ROS nodes to reuse the encoded expertise.

Fig. 2 depicts the results of both transformation activities: First the M2M transformation (1) eliminates the hierarchical

[1]http://freemarker.org/

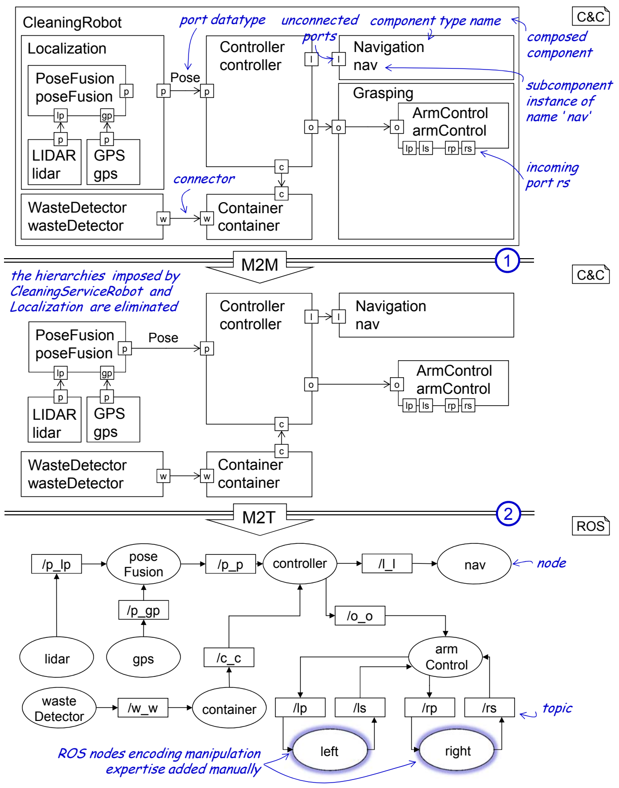

Fig. 2. C&C architecture of a cleaning robot with two arms (top), after applying the M2M transformations for hierarchy elimination (middle), and after applying the M2T transformations producing ROS nodes and topics (bottom).

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.

III. Preliminaries

The presented infrastructure for modular model-driven development of robotics architectures relies on the MontiArcAutomaton C&C ADL, its code generation infrastructure, and the MATrans transformation language generated from MontiArcAutomaton.

A. *MontiArcAutomaton*

MontiArcAutomaton [18] is a modeling infrastructure for software architectures with exchangeable component behavior

DSLs. It comprises the textual MontiArcAutomaton C&C ADL [19], which enables modeling architectures as hierarchies of components, and has been applied in industrial projects [21] and academic robotics contexts [20]. Components are connected via unidirectional connectors between the components' stable interfaces of typed ports. The types of ports are defined in UML/P [22] class diagrams (CDs). MontiArcAutomaton models distinguish between component types and instances, supports component configuration, generic type parameters, and inner components. The model of component `Localization` of Fig. 2 is depicted in Lst. 1.

```
1 component Localization {
2   port out Pose p;
3   component PoseFusion; // The names of subcomponent
4   component LIDAR;      // instances are derived
5   component GPS;        // automatically
6   connect lidar.p -> poseFusion.lp;
7   connect gps.p -> poseFusion.gp;
8   // Connector poseFusion.p -> p is derived also
9 }
```

Listing 1. Textual model of the `Localization` component.

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.

B. 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 [23] 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 [24], and without the accidental complexity [25] of general transformation languages.

Lst. 2 sketches the most important modeling elements and properties of MATrans: Names beginning with "$" are schema variables (e.g., l. 1), "$_" is an anonymous schema variable (l. 3), and the replacement operator "`:-`" (ll. 6-7) 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. 4). 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.

```
1 component $source { port out $type $name; }
2 component $target { port in $type $name; }
3 component $_ {
4   component $source $subS;
5   component $target $subT;
6   not [[ connect $subS.$name -> $subT.$name; ]]
7   [[ :- connect $subS.$name -> $subT.$name; ]]
8 }
```

Listing 2. Excerpt of a transformation that automatically connects ports of the same name in MontiArcAutomaton components.

Lst. 2 matches three distinct components (ll. 1-3), two of which have compatible ports (ll. 1-2), and a composed component containing subcomponents of the former (ll. 4-5). The composed component must not contain a connector between the compatible ports of these subcomponents (l. 6) as this transformation introduces it (l. 7).

C. The Robot Operating System (ROS)

ROS [5] 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" [25] and "notational noise" [26] 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 [6] 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.

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

IV. C&C Model Transformations

Model-to-model transformations [27] can facilitate architecture modeling by adjusting architectures to specific requirements imposed by (1) subsequent processing steps; (2) reducing the cognitive load imposed on the modelers; and (3) instrumenting architectures for further analyses. In the following, we present M2M transformations identified useful for modeling MontiArcAutomaton architectures for robotics and used for the code generation process from MontiArcAutomaton to ROS. New transformations for MontiArcAutomaton can be created and added easily as described in [23], [24]. As 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 changed by removing transformation rules or adding new ones.

A. Eliminating Hierarchies

ROS describes software architectures as flat graphs of nodes and topics, whereas MontiArcAutomaton and many other C&C ADLs [28] support describing architectures as hierarchies of components. 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 [25] and notional noise [26] again. Proper pre-processing can support analysis by flattening the architecture prior to code generation, hence facilitating analysis on architecture model level instead.

Flattening architectures requires eliminating composed components. In MontiArcAutomaton, such components contain at least 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.

```
1  component $_ {
2    not [[ port $_ $_ ]]
3    component $interType $inter;
4    connect [[$inter.$iPort :- $atom.$aPort]] -> $_;
5    [[ :- component $atomType $atom; ]]
6  }
7
8  component $interType {
9    port out $portType $iPort;
10   component $atomType $atom;
11   connect $atom.$aPort-> $iPort;
12 }
13
14 component $atomType { port out $portType $aPort; }
15
16 assign { $atom = uniqueName($atomType); }
```

Listing 3. 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 Lst. 3 takes care of the former. It considers three component types: the top-most component of the system architecture (ll. 1-7), 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. 9-13) yields an outgoing port (l. 10), contains a subcomponent of the atomic type (l. 11), and connects that subcomponent's port to its own outgoing port (l. 12). 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. Reimplementing that for every code generator is costly as well.

B. 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 component cannot send and receive messages other than intended, which ultimately reduces development complexity. To cope with this, 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 Lst. 4 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 (l. 3). A similar transformation is applied to outgoing ports.

```
1 port in [[ $type :- $wrapper<$type> ]] $_;
2
3 assign { $wrapper = "Envelope"; }
4 where { $type != $wrapper }
```

Listing 4. 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.

C. 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, 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.

Consequently, the transformation depicted in Lst. 5 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 (cf., l. 4).

```
 1 component $name<$_> ComponentBody $BODY
 2
 3 component $_ {
 4   component [[ $name<$kind> :- $cName ]] $_;
 5   [[ :- component $cName ComponentBody $PLAIN_BODY ]]
 6   not [[ component $cName ComponentBody $PLAIN_BODY ]]
 7 }
 8
 9 assign {
10     $cName = $name + "Of" + $kind;
11     $PLAIN_BODY = replaceGenerics($BODY, $kind);
12 }
```

Listing 5. Replacing the types of subcomponents yielding generics by new component types with the generics eliminated.

D. Automatically Connecting Ports

MontiArcAutomaton provides means to automatically connect ports under specific conditions (such as implicitly connected event ports in AADL [7]) and with specific connectors. 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. The behavior of `autoconnect port` is illustrated in Fig. 2. Here, `CleaningServiceRobot` 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 Fig. 2). Lst. 2 shows the transformation for the `autoconnect port` statement.

In addition, MontiArcAutomaton provides two shortcuts for connectors 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 matching connectors are connected automatically. Thus, a further transformation normalizes this shortcut by replacing these connectors by connectors between their ports. We also support automated integration of inspection infrastructure via M2M transformations. The corresponding transformations add subcomponents as monitors for every composed component and redirects connectors such that they are monitored by this transformation as described in [23].

E. 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. Omitting superfluous names reduces notational noise [26] and supports developers in focusing on 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 (1) subcomponent instances in case there is just one subcomponent instance for a type; (2) ports if there is just one port of the corresponding type. For instance, `component PoseFusion;` (cf., l. 3 of Lst. 1) will be assigned the derived name `poseFusion`, which can even be used in the model without being made explicit, e.g., for connectors (cf., 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. Lst. 6 depicts a realization of the transformation to add subcomponent instance names using MATrans. Lst. 7 depicts the transformation for incoming ports. Both use the auxiliary method `uncapitalize()` to derive names in their `where` blocks.

```
$SC [[ component $type [[ :- $name ]]; ]]
assign { $name = uncapitalize($type); }
where  { $SC.getInstances().isEmpty() }
```

Listing 6. Deriving default names for subcomponent instances.

```
port $PL [[ in $type [[ :- $name ] ];
assign { $name = uncapitalize($type); }
where { $PL.getName().isEmpty() }
```

Listing 7. Deriving default names for incoming ports.

```
component $_ {
  component $typeName { /* ... */ }
  not [[ component $typeName $_; ]]
  [[ :- component $typeName $instanceName; ]]
}
assign { instanceName = uncapitalize($typeName); }
```

Listing 8. Instantiation of inner components.

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. Lst. 8 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.

V. Transforming C&C 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 a tremendous challenge 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 ROS.

A. 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 in common ADL modeling elements that originates from their common background of component-based software engineering [29]. This includes providing modeling elements for components, connectors, and configurations. Preserving the semantics and properties of the modeling elements in M2T transformations is a tremendous challenge. Especially, when the ADL design concepts differ significantly from the concepts of the targeted middleware. The required M2T transformations are often tied to complex operations and extraordinary effort. Code generators implement such M2T transformations and represent a systematic approach for automated M2T transformations. But code generators are tailored for a specific middleware and are usually 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 the Java implementation as well as ROS support flat graphs only. 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 preprocessing M2M transformations must ensure that the output model conforms to the underlying ADL again.

B. Transforming C&C Architectures to ROS

The prime concerns of transforming C&C architectures to ROS are translating the atomic and composed component types as well as the data types used for ports and creating the configuration files required by ROS projects. This section describes how we addressed these concerns with a code generator using the FreeMarker template engine[3] and the MontiArcAutomaton

[3]http://freemarker.org/

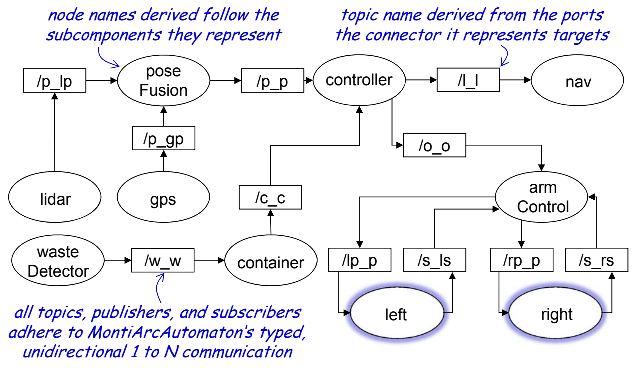
node names derived follow the subcomponents they represent
topic name derived from the ports the connector it represents targets
/p_lp
pose Fusion
/p_p
controller
/l_l
nav
/p_gp
/o_o
lidar
gps
/c_c
arm Control
waste Detector
/w_w
container
/lp_p
/s_ls
/rp_p
/s_rs
left
right
all topics, publishers, and subscribers adhere to MontiArcAutomaton's typed, unidirectional 1 to N communication

Fig. 3. ROS graph of the cleaning robot architecture depicted in Fig. 3.

code generation framework [18]. The intended result of transforming the architecture depicted in Fig. 2 is presented in Fig. 3. Components and connectors are translated to nodes and topics. For instance, the subcomponent `wasteDetector` of type `WasteDetector` is translated to a Python class artifact that defines ROS node and a single publisher to the topic `w_w`, which is derived from the connector between `wasteDetector` and `container`.

MontiArcAutomaton architectures operate in the context of UML/P [22] CDs, hence these must be translated as well. MontiArcAutomaton supports the full expressiveness of UML/P, e.g., interfaces, abstract classes, and generic type parameters. ROS nodes operate in the context of rosmsg models, which do not support these. Consequently, CD 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 translated as expected: primitive attributes becomes rosmsg properties, attributes of complex types become nested properties, 1-to-n relations become arrays of variable length.

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 the paradigm of CBSE 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. 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 Python nodes is not feasible. Instead, each MontiArcAutomaton component type becomes

TABLE I
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

a python class with generic publish and subscribe mechanisms, which initializes a single node, defines a single publisher for each outgoing port, 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. 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 node names, connected topics, and available parameters is generated into roslaunch[4] configuration files. These files orchestrate initialization of ROS graphs and hence are suitable to enact the role of the architecture's top-level component. These are also generated from the MontiArcAutomaton architecture and take care of instantiating the 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.

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).

The resulting code generator enables translating MontiArcAutomaton architecture models to ROS Python nodes. We implemented a similar code generator to translate MontiArcAutomaton components into plain Java artifacts [20]. 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 I, 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 15% bigger.

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.

VI. 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 [14] or DiaSpec [13]. 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 [30] 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 [5] nodes or Orocos [3] components. For instance, the DiaSpec [13] infrastructure comprises an ADL with different component kinds, but does neither support exchangeable model transformations nor exchangeable code generators. The SmartSoft [14] 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 [31] propose modeling self-adaptive software with components and translating it to Fractal [32] component implementations that neither supports extensible M2M, nor exchanging the code generators. RobotML [33] 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.

VII. Conclusion

We have presented a method for separating the concerns of architecture modelers from the technical concerns of code generator developers via domain-specific M2M transformations. The method relies on gradually transforming the architecture under development into representations better processable by code generators. This reduces the effort of creating code generators to interface with specific middleware artifacts and

[4]http://wiki.ros.org/roslaunch/XML

[5]https://eclipse.org/papyrus/

[6]http://www.eclipse.org/acceleo/

ultimately can facilitate model-driven development of robotics architectures by enabling to reuse expertise encoded in middleware artifacts, such as ROS nodes. We applied this concept to the translation of MontiArcAutomaton components into ROS Python nodes. To this effect, we presented the reusable M2M translations we employed 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 believe, this separation can produce better extensible MDD tool chains in robotics and, hence, ultimately facilitate MDD 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.

[2] 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.

[3] 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.

[4] 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.

[5] 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.

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

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

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

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

[10] J. Whittle, J. Hutchinson, and M. Rouncefield, "The State of Practice in Model-Driven Engineering," *Software, IEEE*, vol. 31, no. 3, pp. 79–85, 2014.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

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

[20] ——, "From Software Architecture Structure and Behavior Modeling to Implementations of Cyber-Physical Systems," in *Software Engineering 2013 Workshopband*, 2013.

[21] 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.

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

[23] 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.

[24] 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.

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

[26] D. S. Wile, "Supporting the DSL Spectrum," *Computing and Information Technology*, 2001.

[27] 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.

[28] 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.

[29] 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.

[30] 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].

[31] 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.

[32] 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.

[33] 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.