Tagging Model Properties for Flexible Communication

Manuela Dalibor, Nico Jansen, Jörg Christian Kirchhof, Bernhard Rumpe, David Schmalzing, Andreas Wortmann
Software Engineering, RWTH Aachen University, http://www.se-rwth.de/

***Abstract*—Model-based systems engineering and digital manufacturing aim to facilitate monitoring, integration, and optimization of cyber-physical production systems (CPPS) through so-called "digital shadows". In contrast to "digital twins", digital shadows are purposefully abstracted models of information emitted by the underlying CPPS, hence they do not manipulate the system themselves. We present a method to derive digital shadows from design-time models that can be extended with sophisticated analyses and operate physically distributed without changing the original models. To this end, tag models assign communication information to properties of design-time models from which we generate an Message Queuing Telemetry Transport (MQTT) based communication infrastructure that makes these accessible to other models. This enables the flexible integration and exchange of model information at runtime without polluting these with extra communication information. We present a tagging language for model communication description, a systematic method to apply this to design-time models, generation of a communication infrastructure, and their implementations with the MontiCore language workbench. This, ultimately, facilitates engineering physically distributed digital shadows and, hence, facilitates developing the interconnected CPPS of the future.**

***Index Terms*—Digital Shadow, Digital Twin, Cyber-Physical Systems, IoT, Model-based Systems Engineering**

I. Introduction

Manufacturing is shifting from document-based to model-based approaches to manage the complexity of the future's resilient, distributed, and interconnected cyber-physical production systems (CPPS) [1]. Model-based systems engineering [2] enables this shift by leveraging heterogeneous models as primary development artifacts that conform to domain-specific languages (DSLs) aimed at the different systems engineering stakeholders [3]. Part of this evolution is augmenting CPPS for the industrial internet of things (IoT) with "digital twins" [4] or "digital shadows" [5], [6]. Both serve as abstraction from the underlying CPPS to analyze manufacturing resources and processes. In contrast to digital twins, digital shadows are purposefully abstracted sets of related models based on information emitted by the underlying CPPS, hence they cannot manipulate the system themselves. This prevents automatically interfering with manufacturing processes and resources in hazardous ways.

This research has partly received funding from the German Research Foundation (DFG) under grant no. EXC 2023 / 390621612. The responsibility for the content of this publication is with the authors.

The efficient model-based systems engineering of digital shadows for CPPS demands translation from their prescriptive design models to run-time infrastructure that enables integrating mechanisms for abstraction and analysis as well as physically distributed communication. Where current research focuses on integrating models and model interfaces, these usually only support logical integration [7], provide plain service interfaces without communication infrastructures [8], [9], or pollute the design models with communication information complicating their reuse in other contexts [10]. Moreover, such contributions often focus on model-interpretation [11], [12] infrastructure not easily available on major digital shadow platforms (*e.g.,* Microsoft's Azure, Amazon's Greengrass, or Siemens' MindSphere).

We present a method to derive digital shadow implementations from design models that can be extended with means for sophisticated analyses or abstractions and can operate physically distributed without changing the design models. Instead of pairwise combining design models and communication models a priori or polluting models with communication information, we propose to derive and use specific tagging languages that support adding communication information a posteriori and decoupled from the models' primary concerns.

We, therefore, conceived a language-agnostic method to lift design models to extensible, physically distributed runtime implementations by tagging their inputs and outputs with communication information and making these accessible to other models by integrating these over the Message Queuing Telemetry Transport (MQTT) protocol. This enables the flexible integration and exchange of model information at runtime without adding communication information to these. We, thus, present mechanisms to derive domain-specific tagging languages (DSTLs) specific to the DSLs of communicating models that describe communication information and a systematic method to apply this to design models to make these usable at runtime. The tagging languages are strongly typed w.r.t. the tagged DSLs and the communication infrastructure. Our mechanism supports tagging models elements, checking the validity of the communication description relative to both, and synthesizing a communication infrastructure.

The contributions of this article, hence, are

- A concept to derive DSTLs enriching design DSL models with communication information.

Copyright © 2019 for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

CEUR-WS.org/Vol-2442/paper6.pdf

- A method of deriving digital shadows from design-time models that supports extension with analyses and connects digital shadows via MQTT-based communication using these domain-specific tagging languages.
- A MQTT-based communication infrastructure and model interfaces for tagged models.

This, ultimately, facilitates engineering physically distributed digital shadows and, hence, facilitates developing the interconnected CPPS of the future.

In the following Sect. II introduces preliminaries and Sect. III presents an example. Sect. IV presents our method for tagging-based model communication, before Sect. V discusses it. Sect. VI highlights related work and Sect. VII concludes.

II. Background

We realize a flexible communication infrastructure by integrating model information using models at runtime, the MontiCore language workbench, and tagging languages.

A. Models at Runtime

Traditional model-driven development (MDD) [13] incorporates modeling software systems at design time to foster their development. Runtime models extend this approach to running applications, enabling models to provide an abstract representation of system components [14]. The resulting abstraction combines the static structure of the software with runtime data in a runtime model [15], which evolves with the system [16]. This enables the encapsulation of running systems through runtime models, facilitating maintenance and further development. Additionally, runtime models support monitoring and reconfiguration of software applications and serve as interfaces for self-adapting systems [14]. Thus, they bridge the gap between suitable abstraction and the technical realization during operation.

B. MontiCore Language Workbench

MontiCore [8], [17] is a workbench for developing compositional modeling languages [18]. MontiCore facilitates language definition via context-free grammars (CFGs) for simultaneous development of abstract and concrete syntax. These grammars contain production rules, which consist of terminals and non-terminals to determine the allowed sentences in a language. From a grammar, MontiCore generates an infrastructure for language development comprising parsers, abstract syntax classes, validity checks, and code generators. Textual models that adhere to the underlying language are processed and transformed into a corresponding abstract syntax tree (AST), a structural representation of the model's information without syntactic sugars. MontiCore then performs well-formedness and validity checks on these ASTs using Java context conditions. Finally, template based code generators further process the ASTs and produce artifacts of a target language. For more sophisticated development, MontiCore also supports language composition. This facilitates reusability of constituents of existing languages via inheritance, aggregation, or embedding. Furthermore, it is possible to extend generated abstract syntax classes using MontiCore's TOP mechanism. It allows inserting a handwritten implementation, which is seamlessly integrated into the generated artifacts.

C. Tagging Modeling Languages

DSTLs [10] are a common approach for enriching models with further information without modifying the corresponding DSL definition. They are used to tag constituents of a model for identifying specific topics and attaching additional information for further use (*e.g.,* for code generation). The original models remain free of modifications, which fosters the separation of concerns and thus reusability [10]. In general, DSTLs comprise a tag language `Tag L` and schema `Schema L` to enable tagging for a language `L`. Using language inheritance, `Tag L` extends `L` to access the abstract syntax elements, *i.e.,* its nonterminals. Furthermore, it extends `Common Tag`, a predefined language, which provides default modeling rules to tag models. Thus, `Tag L` combines the original modeling language `L` and `Tag L` to enable tagging for `L`. `Schema L` contains the overall tagging schema for `L`. It defines derivation rules that enable addressing model elements of the original language. Hence, it defines the general set of usable tags concerning their corresponding elements. Model schemas, which conform to `Schema L`, then specify the concrete set of tags. A tag model, which is a model of `Tag L`, finally utilizes this schema to tag elements for models of the original language `L`, optionally adding information. Via this mechanism, the tag model enriches the initial model for a specific purpose while maintaining its general usage.

III. Example

Industry 4.0 is a prominent example of a domain incorporating complex, interconnected, and physically distributed CPPS that cooperate to achieve a common goal and exchange information at runtime. To provide such an example, we developed a smart factory demonstrator simulating a yogurt factory consisting of multiple CPPSs and mobile robots [1]. Via a web-based user interface customers can order and customize yogurts choosing between different kinds of yogurts, additional fruits, and toppings. The factory consists of multiple, physically distributed production stations specialized to fulfill specific tasks. Production stations are connected via conveyor belts or via mobile robots that drive autonomously between the stations and submit ingredients or transport yogurts.

The yogurt production is coordinated by a manufacturing execution system (MES), which manages production and ensures that the yogurt factory produces yogurts as ordered by the customers. When controlling the execution of the factory, the manufacturing execution system relies on models describing the factory, the yogurt configuration, and the behavior of each robot. To produce the configured yogurt the manufacturing execution system relies on information about the current state of the factory and its subsystems. We call these purpose oriented data sets digital shadows. Fig. 1 gives

[1] https://www.youtube.com/watch?v=KTr_uJ5F03E

an overview about the use case. A digital shadow is abstracted information emitted by an underlying CPPS, which provide services and have a location. The digital shadow of the factory is a map that represents the positions of all production stations, available paths between stations, and current locations of the mobile robots. The digital shadow of a transport robot contains its current position, its current task, and its maximal workload. The controller relies on the information provided by the digital shadows to decide which robot should perform which task.

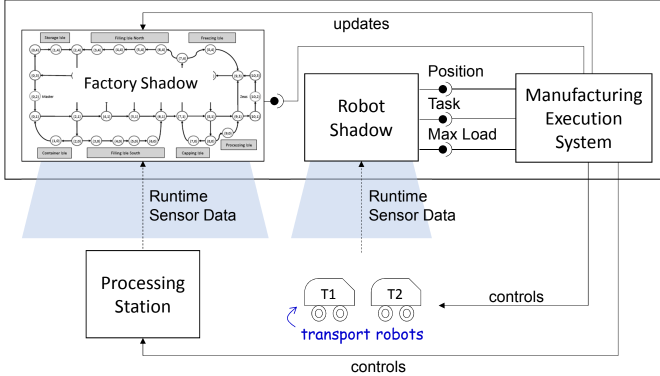

Fig. 1. Digital shadows and their relations in a smart factory demonstrator.

Hardware in industrial settings is often subject to harsh environmental conditions such as *e.g.,* vibration or high humidity levels, that may cause the devices to malfunction [19]. In our scenario, the mobile transport robots are a common cause of failure. They receive requests from different productions stations to transport goods between them, but are subject to inaccurate sensors and hardware failures and thus may lose their way or fail otherwise. If one of the mobile robots fails, it should be ensured that the factory continues to work nonetheless. The MES achieves this by analyzing the digital shadows and automatically detects non-responding robots, by periodically checking registered devices. If the MES registers a non-responding device it has to adapt the factory's behavior accordingly. Digital shadows further help to integrate previously unknown production systems or mobile robots into the system at runtime.

A wide variety of systems are in use in the factory, the development of which requires the experience of different domain experts, design models of various domain-specific languages are used to specify the individual systems. To enable their developers, who may not have MQTT expertise, to exchange information between the individual, physically distributed subsystems of the factory, we developed a methodology that facilitates enriching design models with communication information.

IV. Methodology

We present a method for deriving digital shadow implementations from design-time models to support efficient model-based systems engineering. Derived implementations enable integration at runtime of physically distributed systems through communication over well-defined interfaces. Derivation in this context is the systematic development and generation of an implementation, as well as the integration with the implementation of the corresponding CPPS, using a given set of predefined steps and rules. We use model tagging [10] to identify those elements of a model that should be available to or are expected from other systems in the environment. Therefore, tags define a model's communication interface over which integration and message exchange is possible at runtime.

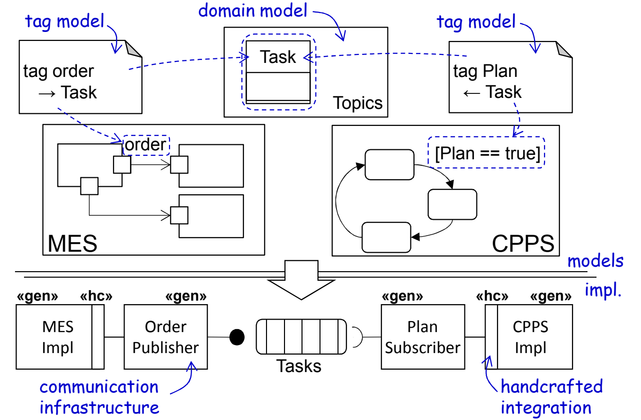

Fig. 2. Enriching models with communication information through tag models and deriving physically distributed run-time implementations.

An exemplary overview of the process for enriching design models with communication information and deriving an implementation for distributed communication is presented in Fig. 2. The top part shows models of our example (see Sect. III), namely an `MES` coordinating production and a `CPPS` fulfilling production tasks. The bottom part shows the derived implementation. The models of `MES` and `CPPS` have no relation to each other and can be reused independently. In particular, knowledge about the systems their implementations interact with at some future runtime is not required at the time of their creation, which enables to deploy them in different contexts. Instead, we define their interfaces ad-hoc in separate tag models, which identify the inputs and outputs to given design models. Communication is realized over the so defined interfaces via topic channels whose types are explicitly modeled in an overlying domain model. To this end, tag models also identify the corresponding topics for inputs and outputs. The exemplary tag models in Fig. 2 identifies the elements `order` of the `MES` and `Plan` of the `CPPS` as output and input respectively and `Task` as the corresponding topic channel. From these models, we derive an implementation that enables integration at runtime and communication of distributed systems using publish/subscribe via MQTT. To this end, we generate publish and subscribe components for outputs and inputs, which realize communication via corresponding topic channels and need to be manually integrated with the implementations of tagged models via lightweight interfaces through the extension of corresponding data classes.

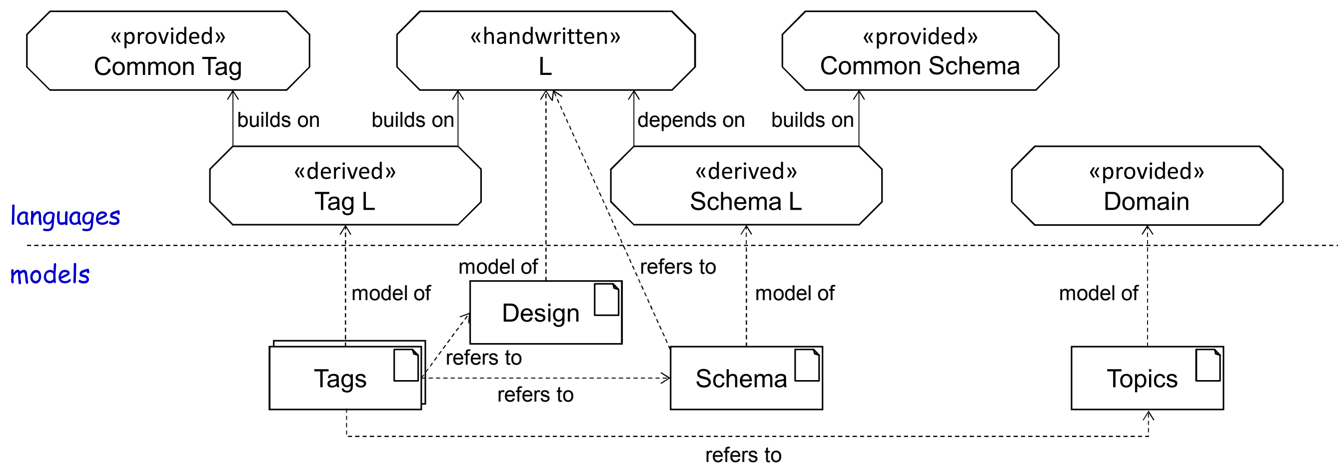

Fig. 3. Overview of the languages and models involved in the derivation of a communication infrastructure, as well as their relations, which is an adapted version of [10] for the purpose of enriching models with communication information.

A. Deriving Tag and Schema Languages

To derive an implementation realizing a flexible communication infrastructure for model integration at runtime via tagging, first, an appropriate tag language suitable for the target modeling language is needed, so that tags can reference elements of models of the target language and enrich them with communication information. Our methodology supports deriving such tag languages and provides the necessary infrastructure for the language derivation process. Fig. 3, adapted from the original source [10], shows the languages and models involved, as well as their relations.

Input for the language derivation process is an existing DSL `L` for which a language engineer wants to enable domain experts to enrich models of `L` with communication information. Given `L` as input, following the language derivation process, a tag language (`Tag L`) and a schema language (`Schema L`) specific to input language `L` are derived. The tag language enables to create tag models (`Tags`) that enrich models of `L` (`Design`) with communication information, and the schema language enables to create tagschema (`Schema`) that define viable tag types. Tags must conform to a tagschema and refer to elements of the input model. To this end, the tag language builds on `L` using MontiCore's language inheritance mechanism. Furthermore, the tag language also builds on a common tag language (`Common Tag`), a provided language that serves as a basis for all tag languages and defines the fundamental structure of all tag models in this context. Besides defining the basic abstract and concrete syntax of all tag models, the grammar of the common tag language also provides extension points through interfaces, which need to be implemented to derive a specific tag language, so that elements of models of the input language can be unambiguously referenced. Fig. 4 shows the MontiCore grammar of the common tag language (`Common Tag`), which is based on [10] with some adjustments for deriving tag languages suitable for enriching target domain models with communication information. Defined in the top level production `TagModel`, each tag model starts with a `conforms to` keywords followed by a list of `QualifiedName` referencing the tagschemata a tagmodel conforms to. Each tag model has a name, which is defined by `Name` after the `tags` keyword, and defines tags for a specific target model, which is referenced by a `QualifiedName` after the `for` keyword. In the following block individual tags can be specified through the `TargetElement` production. Each tag starts with the keyword `tag` followed by a list of `ModelElementIdentifier`, which identify elements of the target model the corresponding tag refers to and is an interface which is implemented in the derived tag language (`Tag L`) by productions of the input language `L`. A `ModelElementIdentifier` uniquely identifies an element and could, *e.g.*, be a `QualifiedName` or the concrete syntax of the target element.

By default, using the derived tag language, tags can be defined for all nonterminals of input language `L`. However, tags must conform to tag types defined in a tagschemata (`Schema`), restricting the element types a tag can be added to and providing a type system for tags. Tagschemata are models of the derived schema language (`Schema L`), which builds on a common schema language (`Common Schema`) and depends on `L` during the language derivation process. Similar to the common tag language, the common schema language is the basis of all schema languages, that is it defines the fundamental structure of all schema models which (`Schema L`) inherits. Besides defining the basic concrete and abstract syntax of all schema models, the grammar of the common schema language also offers extension points through interfaces, which must be implemented so that elements of the input language can be referenced unambiguously. Fig. 5 shows the MontiCore grammar of the common schema language (`Common Schema`). Defined in the `TagSchema` production, a tagschema starts with the keyword `tagschema` and has a name. A `TagSchema` holds a list of tag type definitions

```
01 grammar CommonTags extends Common {                         MCG
02   TagModel =
03     "conforms" "to"          identifier of tagschemas
04      QualifiedName ("," QualifiedName)* ";"
05     "tags" Name "for" targetModel:QualifiedName
06     "{" (Context | tags:TargetElement)* "}";
07                               target model identifier
08   Context = "within" ModelElementIdentifier "{"
09     (Context | tags:TargetElement)* "}";
10                             context for nested
11   interface Tag;            model elements
12   interface ModelElementIdentifier;
13
14   TargetElement =
15         "tag" ModelElementIdentifier
16         ("," ModelElementIdentifier)*
17         "with" Tag ("," Tag)* ";";     topic identifier
18
19   OutTag implements Tag = "outgoing" topic:Name;
20   InTag implements Tag = "incoming" topic: Name;
21 }
```

Fig. 4. MontiCore grammar of the common tag language that provides the fundamental structure of all tag models for enriching domain models with communication information.

```
01 statechart CPPS {                                         MCM
02   initial state Idle;
03   state Assign;                     guard
04   Idle -> Assign [Plan == true] / {plan()};
05   ...  transition                action
06 }
```

```
                                               tag model
01 conforms to SCPubSubTagSchema;                            MCM
02 tags MyTags for CPPS {          transition identifier
03                                 (provides context)
04   within [Idle -> Assign [Plan==true] / {plan()}];
05     tag Plan with incoming Task;
06
07   tag Assigned with outgoing Busy;   topic
08 }           tagged element
```

```
                                  schema model
01 tagschema SCPubSubTagSchema {                              MCM
02   tagtype incoming for Variable;
03   tagtype outgoing for State;
04 }                                 language element
```

Fig. 6. Example for tagging a model with communication information, including an excerpt of a textual statechart, tags that enrich the statechart with communication information, and a tagschema defining tag types the tags must conform to.

(TagType), which are each introduced through the tagtype keyword. The OutTagType and InTagType productions are the implementations of TagType. Each tag type has an optional scope (Scope), which is a list of identifiers (ScopeIdentifier) referencing those language elements of L a tag type can be defined for. The ScopeIdentifier is an interface that is implemented in the schema language (Schema L) during the language derivation process. Here, the name of each nonterminal of L is used as its unique identifier.

```
01 grammar CommonTagSchema extends Common {                   MCG
02   TagSchema = "tagschema" Name
03    "{" TagType* "}";   list of tag types
04
05   interface TagType;
06   interface ScopeIdentifier;
07                                    applicable language
08   Scope = "for"                    element identifier
09     ScopeIdentifier ("," ScopeIdentifier)*;
10
11   OutTagType implements TagType =
12     "tagtype" "outgoing" Scope? ";";
13
14   InTagType implements TagType =
15     "tagtype" "incoming" Scope? ";";
16 }
```

Fig. 5. MontiCore grammar of the common schema language that provides the fundamental structure of all tagschemata in this context.

B. Enriching DSL Models with Communication Information

The derivation of the tag language (Tag L) and schema language (Schema L), which extend the respective commons, enables to enrich input models with communication information, from which we derive implementations that emit digital shadows via MQTT-based communication. Central for the derivation are tag models (Tags) that enrich target models with communication information. Each tag refers to elements of a target model and assigns tag-type-specific properties to that element. Tags of the derived tag language (Tags L) can mark elements of a target model as output or input and assign topics (cf. Fig. 4, ll. 19-20), defining for which model elements and via which topics values are communicated to and expected from systems in the environment. Tags must conform to a tag type definition. By defining tag types in a tagschema (Schema), a language developer specifies the type of model elements a tag can be applied for. To this end, tag type definitions in a tagschema refer to nonterminals in L. Besides elements of the target design model, tags also refer to elements of an overlaying domain model (Topics) specifying the topic channels over which models can communicate. Tags define the model's interface, that is the set of input and output model elements and the topic channel over which their values are communicated. This way of defining interfaces of models facilitates avoiding pollution of the original model, as tags are defined externally. Moreover, it supports the concept of encapsulation, as inter-communication is realized through well-defined interfaces.

An example for enriching a model, here a statechart, with communication information is shown in Fig. 6. The statechart in textual notation is the modeled behavior of a CPPS that can be commissioned to execute some tasks. The CPPS is initially in an Idle state. If a new task is available for the CPPS, then it transitions into a Assign state and plans the execution of its task. The information whether a new task is available may not necessarily be provided by the CPPS itself, but instead is expected as input from other systems in the

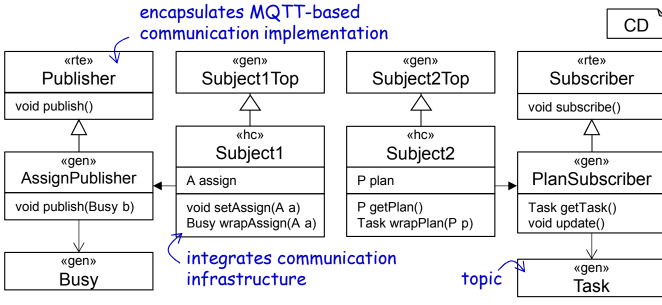

Fig. 7. Implementation for distributed communication derived from enriching design models with communication information using the example shown in Fig. 6.

CPPS's environment, which operate physically distributed and may be exchanged at runtime. To not pollute the behavioral model of the CPPS with additional information and therefore allow to reuse the model in different context, the interface for distributed communication of the CPPS is modeled in the tag model `MyTags`. The tag model refers to the statechart of the CPPS and provides two of its elements with additional properties. That the availability of new tasks is expected as input is provided by the tag of `Plan` (l. 5), which also specifies `Task` as the corresponding topic channel via which the information is received at runtime. To uniquely identify `Plan`, which is not a top-level element of the statechart, its scope needs to be provided. To this end, the `within` statement (l. 4) provides a unique scope identifier through the concrete syntax of the corresponding transition. The tag model further specifies the state `Assign` as output (l. 7) and `Busy` as the corresponding topic channel, that is the CPPS informs systems in its environment whenever it is assigned a task, *i.e.,* enters the `Assgin` state. The corresponding topics themselves are modeled in an overlaying domain model which is not provided here. As stated in the tag model (l. 1), tags must conform to the tagschema `SCPubSubTagSchema`, which defines tag types for statecharts. Here it is specified that only `Variables` of a statechart can be specified as input and that only `States` can be specified as output. Tag models that claim conformity to this tagschema but violate at least one of its tag type definitions are invalid.

C. Deriving a Communication Integration

The integration of systems always poses a challenge, especially if their integration was not planned in advance. Using tags we are able to enrich design models with communication information a posterior. We now need to derive an implementation from these while maintaining the abstraction of communicational detail. As the realization of elements of the base model and the messaging structure to a given topic may have incompatible types, the derivation of the implementation is not fully automated but requires application developers to implement a wrapper between these.

The resulting implementation for the derivation process for MQTT-based integration is shown in Fig. 7 using the example in Fig. 6. From an overlaying domain model, which defines available topics and their message types, we generate classes that type the data structure for the respective topics. In our example, these are the classes `Busy` and `Task`. The derived implementation automatically supports publishing and subscribing to these topics. To this end, for specified outputs and inputs we generate corresponding tag-specific publisher and subscriber classes, respectively, that provide the required functionality for publishing and subscribing to topics with respect to the defined data structure. As `Assign` and `Plan` are specified output and input in our example, we receive generated classes `AssignPublisher` and `PlanSubscriber`. Tag-specific publisher and subscriber classes extend classes `Publisher` and `Subscriber` of the provided run-time environment, which encapsulate a realization of MQTT. Application developers need to integrate this realization into the implementation derived from the corresponding target model. To this end, tag-specific publisher and subscriber classes function as a lightweight interface between the realization of MQTT and the system's implementation. That is, tag-specific publisher classes provide a method `publish(X x)`, where `X` is the corresponding topic data class, such as `publish(Busy b)` of `AssignPublisher`, that automatically wraps received values in an appropriate message and publishes to the corresponding topic. Application developers need to realize the call of this method in the implementation of the target model. To this end, we assume that the generator translating the target model supports the TOP mechanism [8] and provides a data class for each model element that can be enriched with communication information, together with corresponding getter and setter. Using the TOP mechanism, an application developer can extend generated implementation classes such as `Subject1` with handwritten code and thus override the setter `setAssign(A a)` to publish value changes through the call of `publish(Busy b)` of `AssignPublisher` if the setter is called. An application developer furthermore needs to provide a transformation between system-specific and topic specific data types, such as between the implementation of `Assign` and `Busy`. Realizing the data transformation and appropriately calling the corresponding `publish()` method completes the integration of the publishing functionality via the MQTT client. Vice versa, for subscription functionalities an application developer needs to integrate the tag-specific subscriber class, such as `PlanSubscriber`. To this end, the subscriber class stores values received via the MQTT client, which can be access in the implementation of the target model. Again, a transformation between system specific and topic specific data types are required.

Our process for MQTT-based integration of physically distributed systems thus enables flexible integration of modeled systems for arbitrary DSLs. The use of the TOP mechanism further supports easy integration of analyses and data preparation already at the CPPS, *i.e.,* before sending the data. Thus it is possible to modify the transmission behavior, *e.g.,* to define under which conditions a message is sent. While the provided methodology may be applicable to other object-oriented pro-

gramming languages and technological spaces, we realized the derived implementation in Java using M2T transformations via MontiCore's template-based code generators and the Eclipse Paho Java Client [20].

V. Discussion

Our approach of realizing a flexible communication infrastructure for models at run-time is based on tagging languages and the publish/subscribe mechanism of the MQTT protocol. The goal was to achieve this communication without suffering from the disadvantages of current connectivity solutions. The communication structure should support the exchangeability of individual devices. Additionally, the models themselves should be free of communication overhead to facilitate domain experts' development and promote reusability.

Our solution provides flexible communication and communication reconfiguration between models at runtime independent of pre-defined interfaces and underlying hardware. Tagging input and output of the models describes the communication infrastructure. Since the modifications are established in separate tag models, the original models remain unchanged. For communication we use MQTT, additionally increasing the exchangeability of models. The publish/subscribe mechanism allows information transfer between models without the need for mutual knowledge. Thus, both models and communication interfaces are highly flexible and interchangeable. We have decided to use MQTT over other protocols based on the intended use case. MQTT is very well suited for connecting devices in the context of the Internet of Things (IoT). Furthermore, our solution is highly scalable in terms of the number of used models. Following the proposed concept of [10], our approach is extensible to any modeling language, providing a general solution. Since tagging languages can be automatically derived from the modeling language, tagging new languages is convenient. This makes the communication infrastructure highly adaptable to new languages. Since the presented approach is based on tagging languages and the MQTT protocol, it is also subject to their disadvantages. As tagging languages are automatically derived from a base language, the scope identifier suffers limitations, as identifying statements in highly nested structures would require replicating containment hierarchies. Although manual extension of derived tagging languages to provide more convenient scope identifier, such as qualified names, can solve this problem, they may no be available in all cases. Therefore, external tagging is most suitable in rather flat languages. Furthermore, evolution of the base model or modeling language introduce additional effort, as tag model and tagging language need to evolve as well, tough this effort is reduced through the automatic derivation of tagging languages. MQTT requires a broker to distribute messages, which is a bottleneck and single point of failure. However, as our approach is intended for production networks in the IoT, MQTT proved to be very well suited for the realization.

Our contribution provides a suitable approach towards smart factories, as digital shadows can be easily established to support planning, monitoring, and communication of individual factory constituents. The flexible communication infrastructure also supports the application of digital shadows in the context of advanced systems engineering in Industry 4.0. Future work may include the investigation of smart failure management via a business logic over the communication infrastructure.

VI. Related Work

Combining the data of different models at runtime requires monitoring the system. Different commercial cloud providers offer monitoring for their (industrial) IoT devices [21]–[23]. Amazon Web Services (AWS) offers a *device shadow* that is a representation of a device's state at runtime in JSON format [21]. The device shadow can be modified and will then be synchronized with the device the next time it connects to AWS. Microsoft Azure offers similar functionality with *Azure digital twins* [22]. *Spatial intelligence graphs* can further describe the location the devices are deployed in. Similarly, Siemens Mindsphere offers a digital twin, whose data can be used to optimize the performance of the production system [23]. These systems do not include design-time models while extracting knowledge from monitored data. None of these supports deriving digital twins (or shadows) from models.

B-COol [24] allows connecting models using *coordination patterns*. The approach requires all connected languages to conform to an interface that abstracts from the behavioral semantics of the language. Models then communicate via events which are combined using coordination patterns. In contrast to B-COol, our method allows us to connect models distributed over multiple devices. MontiArc [25] allows to embed behavioral DSLs into components of component & connector architectures. Information between models of embedded DSLs is exchanged over component ports and demands for their apriori architectural integration by establishing connectors between the components at design time. The same holds, when using other architecture modeling languages, such as UML-RT [26], for instance. This inflexibility is also visible in many IoT modeling frameworks. Many frameworks, such as MDE4IoT [27], embed communication-related code in the models. SysML4IoT [28] transforms system models into platform specific models that are based on the Data Distribution Service (DDS) to realize a publish/subscribe communication. ThingML [29] and FRASAD [30] provide DSLs that require users to specify sent and received messages. Compared to these approaches, our approach is more flexible as the modeler does neither need to know about the concrete messages, neither do the models rely on a specific communication framework or are polluted with communication information.

UML profiles [31] enable to enrich UML models with stereotypes and tag values through a generic extension mechanism. Though originally limited to UML models, they have been extended to EMF [32]. UML stereotypes and tag values provide additional information to models, however in an informal way, as properties are not made explicit for stereotypes and there is no explicit type system for tagged values. Moreover, they are added to model elements directly whereas

in our approach tags are provided externally to support adding information a posteriori and using models in different context.

VII. Conclusion

We have presented a non-invasive method to enrich design models with communication information from which their MQTT-based integration can be derived. This method relies on tagging base models according to communication-specific tag schemata that define which of their properties are communicated over MQTT topics. From these, we generate the infrastructure to make the models' implementations talk (*i.e.,* read and write messages from and to MQTT topics). This generation considers handcrafted augmentations and automatically integrates these into the resulting artifacts such that some analyses can already be performed at the CPPS before sending tagged information. Overall, our method facilitates deriving digital shadows of system models without polluting the design models and, ultimately, can serve as a basis for self-adaptive digital twins of CPPS.

References

[1] J. A. Estefan, "Survey of Model-based Systems Engineering (MBSE) Methodologies," *Incose MBSE Focus Group*, vol. 25, pp. 1–12, 2007.

[2] A. L. Ramos, J. V. Ferreira, and J. Barceló, "Model-based Systems Engineering: An Emerging Approach for Modern Systems," *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, vol. 42, pp. 101–111, 2011.

[3] A. Wortmann, B. Combemale, and O. Barais, "A Systematic Mapping Study on Modeling for Industry 4.0," in *Conference on Model Driven Engineering Languages and Systems (MODELS'17)*. IEEE, September 2017, pp. 281–291.

[4] B. Schleich, N. Anwer, L. Mathieu, and S. Wartzack, "Shaping the Digital Twin for Design and Production Engineering," *CIRP Annals*, vol. 66, pp. 141–144, 2017.

[5] T. H.-J. Uhlemann, C. Lehmann, and R. Steinhilper, "The digital twin: Realizing the cyber-physical production system for industry 4.0," *Procedia Cirp*, vol. 61, pp. 335–340, 2017.

[6] D. Wohlfeld, V. Weiss, and B. Becker, "Digital shadow–from production to product," in *17. Internationales Stuttgarter Symposium*. Springer, 2017, pp. 783–794.

[7] A. Haber, M. Look, P. Mir Seyed Nazari, A. Navarro Perez, B. Rumpe, S. Voelkel, and A. Wortmann, "Integration of Heterogeneous Modeling Languages via Extensible and Composable Language Components," in *Proceedings of the 3rd International Conference on Model-Driven Engineering and Software Development*. Scitepress, 2015.

[8] K. Hölldobler and B. Rumpe, *The MontiCore Language Workbench, Edition 2017*. Shaker, 2017.

[9] M. E. V. Larsen, J. Deantoni, B. Combemale, and F. Mallet, "A behavioral coordination operator language (bcool)," in *2015 ACM/IEEE 18th International Conference on Model Driven Engineering Languages and Systems (MODELS)*. IEEE, 2015, pp. 186–195.

[10] T. Greifenberg, M. Look, S. Roidl, and B. Rumpe, "Engineering Tagging Languages for DSLs," in *Conference on Model Driven Engineering Languages and Systems (MODELS'15)*. IEEE, 2015, pp. 34–43.

[11] C. Wende, N. Thieme, and S. Zschaler, "A role-based approach towards modular language engineering," in *International Conference on Software Language Engineering*. Springer, 2009, pp. 254–273.

[12] E. Bousse, T. Degueule, D. Vojtisek, T. Mayerhofer, J. Deantoni, and B. Combemale, "Execution Framework of the Gemoc Studio (Tool Demo)," in *International Conference on Software Language Engineering*. ACM, 2016, pp. 84–89.

[13] 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.

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

[15] M. Rohr, M. Boskovic, S. Giesecke, and W. Hasselbring, "Model-driven Development of Self-managing Software Systems," in *Model-Driven Engineering Languages and Systems*. Springer, 2006.

[16] J. Dubus and P. Merle, "Applying OMG D&C Specification and ECA Rules for Autonomous Distributed Component-based Systems," in *International Conference on Model Driven Engineering Languages and Systems*. Springer, 2006, pp. 242–251.

[17] 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*, ser. Communications in Computer and Information Science, vol. 580. Springer, 2015, pp. 45–66.

[18] J.-M. Jézéquel, M. Leduc, O. Barais, T. Mayerhofer, E. Bousse, W. Cazzola, P. Collet, S. Mosser, B. Combemale, T. Degueule, R. Heinrich, M. Strittmatter, J. Kienzle, G. Mussbacher, M. Schöttle, and A. Wortmann, "Concern-oriented language development (COLD): Fostering reuse in language engineering," *Computer Languages, Systems & Structures*, vol. 54, pp. 139 – 155, 2018.

[19] V. C. Gungor and G. P. Hancke, "Industrial Wireless Sensor Networks: Challenges, Design Principles, and Technical Approaches," *IEEE Trans. on Ind. Electronics*, vol. 56, no. 10, pp. 4258–4265, Oct 2009.

[20] "Eclipse Paho Java Client," [Online]. Available: https://www.eclipse.org/paho/clients/java/ Last checked: 25.08.2019.

[21] AWS Documentation, "What Is AWS IoT?" [Online]. Available: https://docs.aws.amazon.com/iot/latest/developerguide/what-is-aws-iot.html, Accessed 11.06.2019.

[22] Microsoft, "Azure Digital Twins," [Online]. Available: https://azure.microsoft.com/en-us/services/digital-twins/, Accessed 11.06.2019.

[23] Siemens, "MindSphere The Cloud-based, Open IoT Operating System for Digital Transformation," [Online]. Available: https://cdn2.hubspot.net/hubfs/1147371/Resources/Siemens_MindSphere_Whitepaper.pdf, Accessed 11.06.2019.

[24] M. E. Vara Larsen, J. DeAntoni, B. Combemale, and F. Mallet, "A Behavioral Coordination Operator Language (BCOoL)," in *ACM/IEEE 18th International Conference on Model Driven Engineering Languages and Systems (MODELS)*, Sep. 2015, pp. 186–195.

[25] A. Butting, A. Haber, L. Hermerschmidt, O. Kautz, B. Rumpe, and A. Wortmann, "Systematic Language Extension Mechanisms for the MontiArc Architecture Description Language," in *European Conference on Modelling Foundations and Applications (ECMFA'17)*, ser. LNCS 10376. Springer, Juli 2017, pp. 53–70.

[26] K. Jahed and J. Dingel, "Enabling Model-Driven Software Development Tools for the Internet of Things," in *11th Workshop on Modelling in Software Engineering (MiSE'2019)*, May 2019.

[27] F. Ciccozzi and R. Spalazzese, "MDE4IoT: Supporting the Internet of Things with Model-Driven Engineering," in *10th International Symposium on Intelligent Distributed Computing*, Oct 2016.

[28] M. Hussein, S. Li, and A. Radermacher, "Model-driven Development of Adaptive IoT Systems," in *Proceedings of MODELS 2017. Workshop ModComp*, vol. 2019. Austin, United States: CEUR, 2017, pp. 17–23.

[29] N. Harrand, F. Fleurey, B. Morin, and K. E. Husa, "ThingML: A Language and Code Generation Framework for Heterogeneous Targets," in *Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems*, ser. MODELS '16. New York, NY, USA: ACM, 2016, pp. 125–135.

[30] X. T. Nguyen, H. T. Tran, H. Baraki, and K. Geihs, "FRASAD: A framework for model-driven IoT Application Development," in *IEEE 2nd World Forum on Internet of Things (WF-IoT)*, 2015, pp. 387–392.

[31] B. Selic, "A Systematic Approach to Domain-specific Language Design using UML," in *10th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC'07)*. IEEE, 2007, pp. 2–9.

[32] P. Langer, K. Wieland, M. Wimmer, J. Cabot *et al.*, "EMF Profiles: A Lightweight Extension Approach for EMF Models." *Journal of Object Technology*, vol. 11, no. 1, pp. 1–29, 2012.