Application Configuration via UML Instance Specifications

Ansgar Radermacher, Shuai Li and Matteo Morelli
CEA, LIST, Laboratory of Model Driven Engineering for Embedded Systems, P. C. 174, Gif-sur-Yvette, 91191, France

Keywords: UML, Instance Modeling, Configuration.

Abstract: One way to design complex systems is to use a model driven approach. Model Driven Engineering (MDE) promotes the use of models are primary artifact for analysis, design and implementation of a system. In this paper, we focus on component-based models including classes (representing components), hierarchical composition (classes with parts) and instance specifications. Instance specifications describe instances of a system, i.e. provide values for (a subset of) the attributes of class attributes. However, the use of instance specifications without additional tool support is tedious, since several references need to be setup. This paper will show some mechanisms (notably tables) to ease the usability. There are also different ways to organise instance specifications which have advantages and inconveniences. This paper lists them and provide hints in which situation a certain variant could be used. Code generation from models needs to take instance configuration into account, but application configuration via instances is still not fully supported by all tools. We will show how code generated from instance specifications can look like. This code becomes more interesting for adaptive applications that need to change between different configurations at runtime.

1 INTRODUCTION

The modeling of systems is often based on composite structures, i.e. a system contains several subsystems or components which in turn can contain further subsystems or components. In such a hierarchical structure, composite classes have *parts* which in turn are typed with other classes. A typical case is a top-level class describing the functional architecture of a system. The system is made-up of several (possibly composite) software components with configuration parameters, which can be set to different values depending on different analysis scenarios the designer wants to study. How can such a system be configured?

In many domains, the configuration is specified using Excel tables and then (more or less) manually reflected in code or configuration files. But it is possible to do the configuration on the model level. This has the advantage of enabling systematic and automated consistency validation of model information, which is taken into account by code generation.

The use of UML mechanisms for configuration purposes is part of an initiative to ease (and if possible automate) modeling tasks. For instance, some configuration data can be computed from model information. If behavior descriptions in the the model are for instance enriched with execution time information, a schedulability analysis can decide whether the system is schedulable and –if yes– automatically calculate suitable task priorities that become part of the system configuration.

Classes in UML have a set of properties that can capture configuration values, either via a default value or an instance specification that describes a possible instance of this class. While UML has a quite complete set of modeling mechanisms for instances, there are different ways of using these which have different advantages and disadvantages. Additional tooling is required to ease the usability of instance specifications.

The organisation of the paper is the following.

Section 2 describes the meta-model of UML instance specifications, their properties and use as described in the standard. Section 3 sketches a simple example application which we want to configure and several ways to combine default values and instance specifications. Section 4 shows tooling support, on the one hand oriented to enhance usability, on the other hand to show the impact of instance specifications on the code generation. Section 5 discusses related work. The conclusion and future work are presented in section 6.

Radermacher, A., Li, S. and Morelli, M.
Application Configuration via UML Instance Specifications.
DOI: 10.5220/0007583104950502
In *Proceedings of the 7th International Conference on Model-Driven Engineering and Software Development (MODELSWARD 2019)*, pages 495-502
ISBN: 978-989-758-358-2
Copyright © 2019 by SCITEPRESS – Science and Technology Publications, Lda. All rights reserved

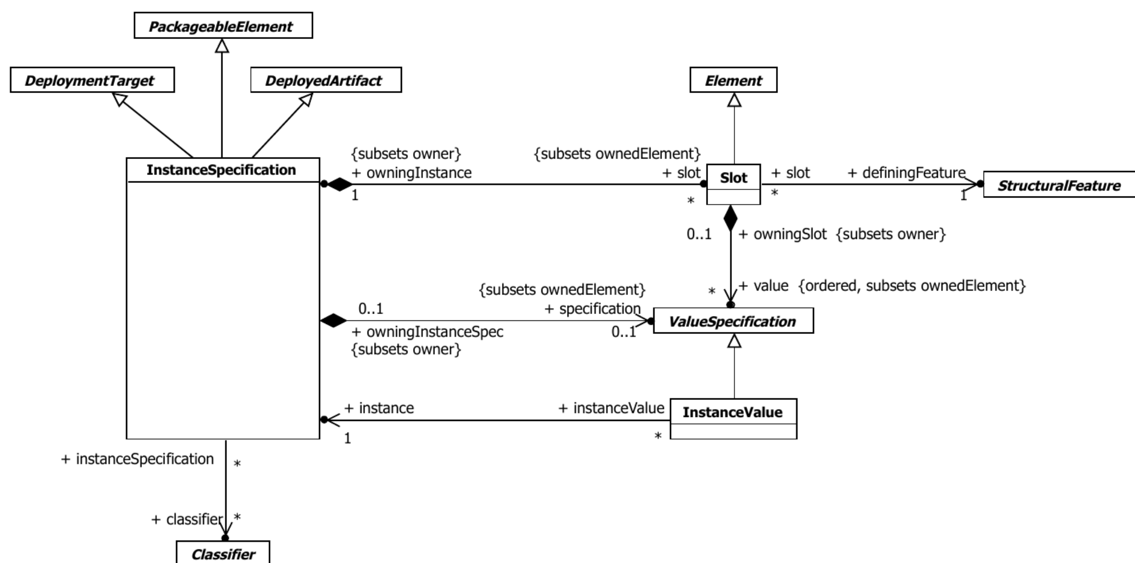

Figure 1: Meta-Model of UML instance specification (UML specification 2.5.1, section 9.8).

2 UML INSTANCE SPECIFICATIONS

In this section, we describe an instance specification in detail. The current UML 2.5.1 standard(OMG, 2017) describes instance specification as follows

> "An InstanceSpecification represents the possible or actual existence of instances in a modeled system and completely or partially describes those instances.
> A Slot specifies that an instance modeled by an InstanceSpecification has a value or values for a specific StructuralFeature, which shall be a StructuralFeature that is related to a classifier of the InstanceSpecification owning the Slot by being a direct attribute, inherited attribute, private attribute in a generalization, or a memberEnd if the classifier is an Association, but excluding redefined StructuralFeatures. The values in a Slot shall conform to the defining StructuralFeature of the Slot (in type, multiplicity, etc.). The values in a Slot are specified using ValueSpecifications."

Figure 1 shows the meta-model definition of UML instance specifications, as in section 9.8 of the UML 2.5.1 specification. An instance specification is part of a package, it can play the role of a deployed artifact (if it defines an artifact) and can also be a deployment target.

An attribute of an instance specification is the *classifier*: a list of classifiers that a specification classifies. In most cases, an instance specification references a single classifier, often a class or data type.

The set of slots describes values for structural features (typically properties of a class or datatype) of the instance. Not all properties of the InstanceSpecification need be represented by Slots, in which case the InstanceSpecification is a partial description. An InstanceSpecification may represent an instance at a point in time (a snapshot). Changes to the instance may be modeled using multiple InstanceSpecification, one for each snapshot. Section 9.8.3 of (OMG, 2017) states

> "It is important to keep in mind that InstanceSpecification is a model element and should not be confused with the instance that it is modeling. As an InstanceSpecification may only partially determine the properties of an instance, there may actually be multiple instances in the modeled system that satisfy the requirements of the InstanceSpecification. On the other hand, an InstanceSpecification may model a situation which is not actually supposed to occur in the modeled system, in which case no instance meeting the requirements of the InstanceSpecification may ever actually occur in the system."

An InstanceValue is a ValueSpecification that references an InstanceSpecification. Any slots in the InstanceSpecification then provide values for the corresponding StructuralFeatures of the instance by evaluating the ValueSpecifications associated with those slots.

Please note that an InstanceValue does not own the InstanceSpecification to which it refers; multiple InstanceValues may refer to the same InstanceSpecification (OMG, 2017).

SCIENCE AND TECHNOLOGY PUBLICATIONS

Instance specifications have to respect a set of constraints, notably:

- An InstanceSpecification can act as a DeployedArtifact if it represents an instance of an Artifact.
- Each slot must reference a different defining feature to avoid conflicting values for this feature
- The defining feature of each slot is a StructuralFeature (directly or inherited) of a classifier of the InstanceSpecification.
- An InstanceSpecification can act as a DeploymentTarget if it represents an instance of a Node and functions as a part in the internal structure of an encompassing Node.

3 EXAMPLES

Let us look at a very simple example of a robotic component. A mapper component has two configuration attributes: a boolean indicating whether it is working indoor or not and a `noOfScans` attribute that indicates the number of scans per second to be done. In UML, the component is modeled as a class with attributes. Figure 2 shows the associated class diagram containing the attributes along with their default values. On the model level, the default values for the two attributes are a LiteralBoolean and a LiteralInteger, respectively. See also the Papyrus4Robotics webpage[1].

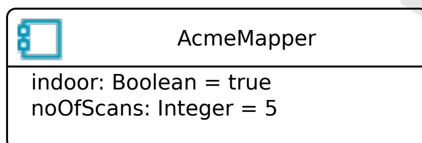

Figure 2: A simple class with two attributes.

Now consider that the component is used in a specific system in which we want to configure one of the values in a different way. Figure 3 shows a “system” class with three parts. Each of the part is typed with a different robotic component. The part `m` is typed with the class AcmeMapper – `m` is thus an attribute with aggregation kind “composite”.

We choose to provide a default value for m that changes the `noOfScans` attributes of the AcmeMapper. Here, the default value is an InstanceValue element that points to a separate instance specification for the AcmeMapper. Unlike the InstanceValue reference, the instance specification cannot be stored within the default value, i.e. the user has to decide (typically following a convention) in which package he wants to put the specification. In our case, we choose to create a separate top-level package `instances` in which we place the instance (and similar ones for the two other components in our system).

[1] Getting started with Papyrus4Robotics: https://robmosys.eu/wiki/baseline:environment_tools:getting_started_with_papyrus4robotics

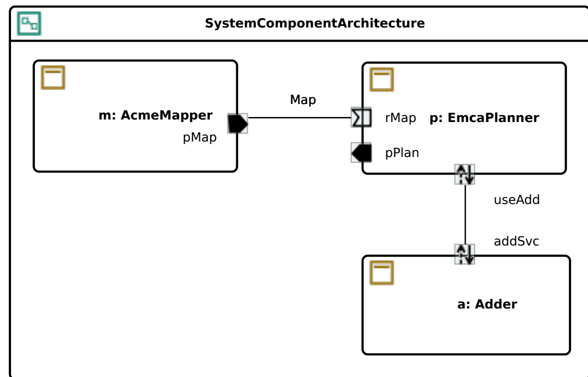

Figure 3: A component assembly by means of a composite class.

Figure 4 shows the instance specification. It assigns a different value (15) to the `noOfScans` attribute, i.e. overrides the default value on class level. Note that this instance specification is partial in the sense that it does not re-define a value for all attributes of the class.

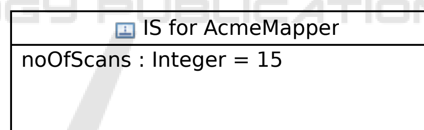

Figure 4: An instance specification for the AcmeMapper class.

The scheme above works fine with a single hierarchy level. In case of nested components, it is also possible to override the default value at multiple levels. If we now assume that class `SystemComponentArchitecture` describes a subsystem in a more generic architecture definition, we can add a default value for the attribute within the generic architecture (container) that describes first the subsystem and then overrides the default value for the part `m` with an additional slot.

In this case, the slots of the top-level instance specification use an InstanceValue that in turn points to an instance specification at the second level. As shown in Figure 5, it is thus possible to describe to override the default value for m with an instance specification that re-assigns the value 10 to the `noOfScans` attribute. Please note, that we only show a subset of

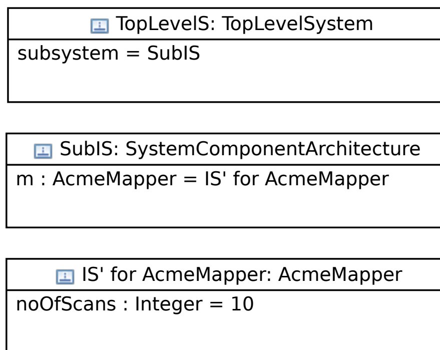

Figure 5: A set of hierarchical instance specifications.

the slot values to simplify the figure.

It is possible that some instances are *shared*. Imagine that an additional component, e.g. the planner is accessed by multiple components. While it is possible to make this explicit via ports and external connections as shown in diagram 3, it is also possible to access a the planner via an internal attribute as a shared component (an attribute with aggregationkind = shared). In case of the example, the planner would be an internal shared attribute within the mapper and adder. Figure 6 shows a variant of the mapper component that accesses the planner via a shared instance, graphically denoted by the dashed line. The motivation for using shared components is to "internalize" commonly used components instead of have a large number of connections in bigger systems that all point to a set of common components. However, it must be unambiguous to which component the shared attributes refer to and that at least one instance exists. The former is not a strict condition: while it becomes ambiguous to which instance a shared attributes refers to when we only take the class composition into account and multiple instances exist, an instance-value in the set of instance specifications would explicitly reference exactly one instance. The tree structure formed as in Figure 5 becomes a graph structure with shared elements. Yet, it might be useful to impose additional rules that avoid the ambiguity already on the class level, either by adding the constraint that there must be exactly one non-shared attribute in the parent composition structure for each shared attribute or setting up precedence rules which reference should be chosen.

Discussion. The nesting of instance specifications is quite difficult to handle for a user, as we will see in section 4. But besides of this aspects, another question is to which extent default values should be used. As we have seen in the previous section, we can always override a default value with an instance specification referenced from a higher level.

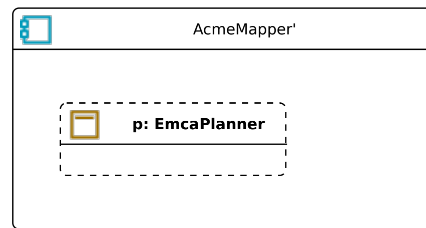

Figure 6: Access to shared instance specifications.

In some cases, we want to reuse an existing component assembly (represented by a composite class such as `SystemComponentArchitecture`) in different contexts. For attributes that are likely to keep their configuration value, the assignment of default values make sense. For others, it is preferably not to use a default value but to require a configuration via an instance specification.

Such a tree of instance specification has been shown in the Figure 5. As already mentioned, it is not possible to reflect the tree structure via a real nesting of instance specifications since a value cannot directly represent a new instance specification but point only indirectly to another instance (via an InstanceValue). While this has been done with the intention to favor reuse of instance specifications, it requires a flat representation of a tree-like structure (of which some elements might be shared). One way of managing the hierarchy is to use a suitable naming convention by encoding for instance the tree path (separated e.g. via ".") in the path, e.g. `topIS.subIS.m`. In order of being effective, tool support is required to update the names whenever part names change or the hierarchy changes.

The possibility of sharing is a powerful modeling mechanism that simplifies the access to commonly used components. But it needs to be used carefully to avoid ambiguities.

The systematic use of (nested) instance specifications for configuration purposes can be compared with an approach that is called *deployment plan* (OMG, 2006) in the context of the CORBA component model: a set of configuration values along with deployment information. In some projects, we've already used a hierarchy of instance specifications to describe the application components. In addition, another set of instance specifications describes deployment targets, i.e. a set of hardware nodes containing configurable attributes such as the amount of RAM or ROM. Allocation information points from instances of software components to instances of hardware nodes.

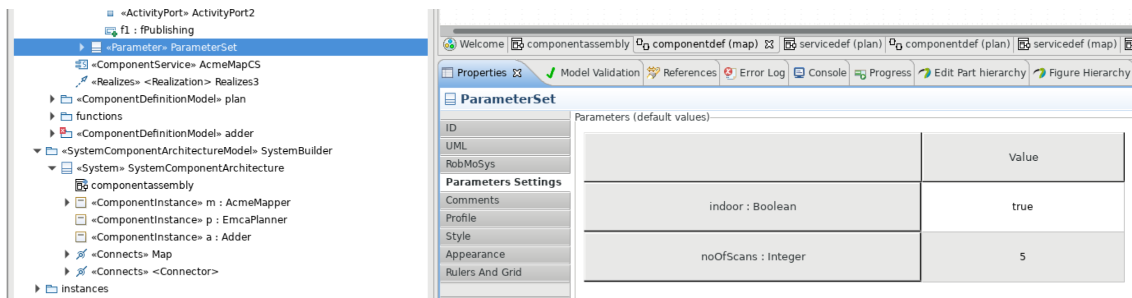

Figure 7: Use a table to configure default values.

4 TOOLING

This section is split into two parts. In the first, we show ways to ease the modeling of default values and associated instances. In the second, we show code generation support from models containing instance specifications.

4.1 Enhanced Usability of Instance Specifications

The creation of instance specifications can be quite tedious, as already sketched in section 3. The user has to specify the referenced classifier and then add a slot for each attribute that needs to be configured. The slot in turn requires the selection of the defining feature reference and a value specification. If the attribute to configure is a primitive type holding a numerical (integer or real), boolean or string value, UML provides appropriate value definitions such as LiteralInteger, LiteralReal, LiteralBoolean or LiteralString. As already mentioned, composite data types require an additional instance specification that is referenced from an InstanceValue. The latter is also used for enumeration literals (which are effectively instance specifications).

Additional tool support is required to ease the creation and visualization of instance values. Figure 7 shows a table that configures a set of default values of a class, specially the `AcmeMapper` introduced in section 3. This table is an extension of the Papyrus[2] UML modeler that has been provided in the context of a customization for the robotics domain (without being actually specific for that domain). When the class to configure is selected, all attributes are shown (optionally only showing attributes that are tagged as configuration attributes). The user can simply enter a value in a string from and the tool will automatically chose a suitable value definition depending on the attribute type. Please note that this table is only an additional notation – the model still contains standard UML instance and value specifications.

Figure 8 shows a quite similar table in case of selecting a part in the component assembly, in this case the part `m` representing an instance of the `AcmeMapper` component. The table shows the configuration attributes of the referenced component. If the part has a default value referencing an instance specification, the table checks whether one of the slots in the instance specification overrides a default value on the class level. An overridden value is shown with a yellow background. If the user edits a value, the associated slot and value will be automatically created. If the user enters a new value and no instance specification exists yet, it will be created first (using the naming convention to place the instance specification in a specific package) and the classifier reference is setup accordingly. If the field value corresponds either to the default value on the class level or is empty, the corresponding slot in the instance specification is removed.

These two tables simplify the configuration via instance specifications considerably. An additional tool support (which is part of the Payprus extension SW designer[3]) manages the automatic creation of a tree-like set of instance specifications from a composite class with parts that are typed with other classes. Figure 9 shows different deployments of an application. Each of these deployments is represented by a package containing a set of generated instance specifications. The naming of the instance specification name reflects the hierarchical decomposition. The user has the possibility to update the names of instance specifications via a command in the context menu. This is required after changes of part names or of the compositions hierarchy.

The instances also contain allocation information. If an instance specification is selected, a table as in Figure 8 is shown. We plan to use a hierarchical table to enable an easy navigation to different levels within

[2] Eclipse Papyrus, see eclipse.org/papyrus

[3] wiki.eclipse.org/Papyrus_Software_Designer

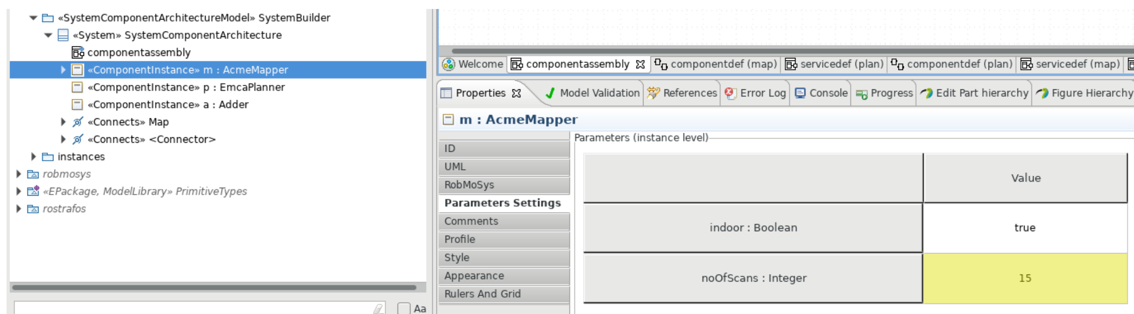

Figure 8: Use a table to configure values for a part typed with a given component.

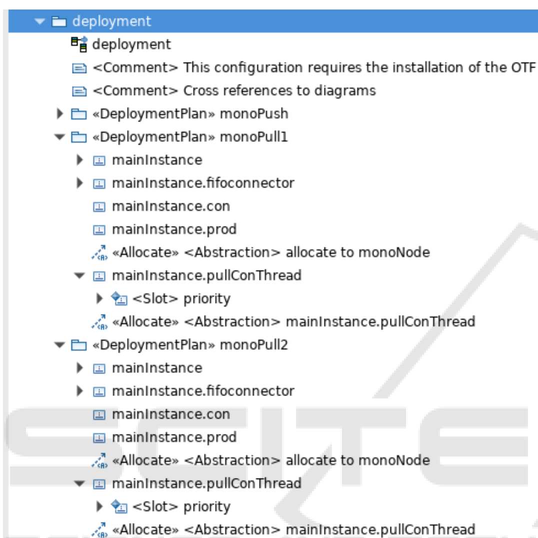

Figure 9: Multiple deployments.

the tree structure.

Compared to the diagrams shown for instance in Figure 5, tables scale much better: a table with hundreds of entries is still quite easy to handle, whereas a diagram should not represent more than a dozen instance specifications. As the relationships between instances are typically tree like (optionally with some sharing) the graphical representation does not provide an added value and (hierarchical) tables are a good representation of the underlying configuration data. Thus, tools change the way a user perceives the instance specification metaclass and its relations (e.g. composition of instance specifications).

4.2 Code Generation from Instance Specifications

The specification of configuration data needs to be reflected in the generated code. In case of Papyrus SW designer, code generation for C++ and Java is supported. Default values are directly initialized in the

Listing 1: Attribute configuration in boot-strap code in C++.

```
void BootLoader::init() {
  // configure attributes
  mainInstance.pullConThread.priority
     = 5;
  mainInstance.fifoconnector.m_size =
     30;
  mainInstance.createConnections();
  ...
```

generated class definitions. In case of a deployment plan, boot-strap code for each deployment target is generated. This code contains the instance configuration. In this case, the configuration of all non-default attributes is done by this top-level boot-strap code. The code in listing 1 shows the attribute configuration for one of the deployments in Figure 9.

This example is interesting, since one of the configuration attributes has a specific semantics – thread priorities could be the result of a scheduling analysis at the model level. The advantage of doing the configuration at the model level is that the relevant information is centralized within the model.

5 RELATED WORK

Instance specifications are a powerful modeling instrument. Their use is not new, but surprisingly little work has been done to describe systematically how these can be use to configure applications. Most work is part of either tutorials and tool descriptions. The tool MagicDraw from NoMagic provides accelerations based on drag & drop to assign a classifier to an instance specification or to assign a default value to a property (NoMagic, 2019a). However, there is no support for automatically creating a set of instance specifications from a composition of classifiers. If a larger number of instance specifications needs to be managed, NoMagic recommends the use instance tables as well to enhance usability, see

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS

(NoMagic, 2019b). Instance tables are also used by another Papyrus extension, the MOKA (CEA, 2019) tool that enables the direct execution of (fUML) models. The documentation for Rhapsody contains information about instance modeling (IBM, 2019), but does not give any hints on how to use these systematically. Rhapsody simplifies the assignment of values via popup dialogs in which values could be entered directly instead of explicitly having to chose the respective UML element (such as LiteralString or LiteralInteger).

The work done in (Agarwal et al., 2000) states that traditional class-based models are not sufficient in case of enterprise models, but need to be accompanied by instance models. The management of these instance models needs to be facilitates by tools. However, the work is not specific for UML models and does not cover the possibilities of UML instance specifications. Whole-part relationships have been examined by (Guizzardi et al., 2002), the paper deals with the aggregation vs. composition aspects between classes and instances. While the paper is specific for UML (the relatively old version 1.4), it does not particularly deal with instances. Sharing of instances is an important concept in the FRACTAL component model (Bruneton et al., 2006).

Many tools generate code from UML models, but most do not mention specific support for generating code from instance models. In case of Rhapsody, default values are either initialized via assignment or in the constructor.

(Ciccozzi et al., 2012) presented a mechanism to support full code generation from UML models. The approach that has been used in the European project CHESS, contains a dedicated model of component and port instances, taking multiplicity information into account. The latter assures that a port of a component instance interacting with others has the matching multiplicity (e.g. two, if the component instance interacts with two others) and to generate the appropriate code. Assuring that multiplicity information matches is the subject of (Mammar and Laleau, 2014) which attacks this issue in a more formal way using the *B* language.

In case of Papyrus SW designer already mentioned in section ref sec:tooling, the support is twofold: it supports the initialization of default values as well as an initialization via boot-strap code that might override the default values. The tool also takes multiplicity information into account and has a validation rules that check whether these are matching. An extension of this tool supporting the (re-) configuration of instances at run-time is presented in (Hussein et al., 2017). A UML state-machines captures the different application states along with transitions that are triggered by adaptation events. Each state is coupled with a deployment plan, i.e. a set of instance specification that correspond to this state.

6 CONCLUSION

In this paper, we have shown how to use UML instance specifications to configure a system. While UML instance specifications are quite powerful, their usage needs additional tool support to improve usability. Tables are a suitable and saleable way to configure instances – and ease the transition for developers that are use to configuration via Excel tables. While not being new, there is surprisingly little work that list different ways to organize these instances, e.g. the use of default values, the possibility override these or the sharing of instance specifications. Putting the application configuration into the model enables a first validation on the model level.

We have also shown that UML lacks the possibility of nesting of instance specification. While instances values can point to other instance specifications, the instance specifications can now own other instance specifications.

Code generation support is required, if the generated code should actually take the configuration values into account. This becomes more interesting, if we take re-configurable applications into account that are able to change between different configurations.

The work underlying this paper has been partially funded by the RobMoSys project from the European Union's Horizon 2020 research and innovation programme under grant agreement No 732410.

REFERENCES

Agarwal, R., Bruno, G., and Torchiano, M. (2000). Instance modeling – beyond object-oriented modeling. In *Proceedings of the 3rd International Conference on Information Technology (CIT 2000)*.

Bruneton, E., Coupaye, T., Leclercq, M., Quéma, V., and Stefani, J.-B. (2006). The FRACTAL component model and its support in Java. *Software-practice and Experience*, 36:1257–1284.

CEA (last visited 2019). *Papyrus MOKA User Guide*. https://wiki.eclipse.org/Papyrus/UserGuide/ModelExecution.

Ciccozzi, F., Cicchetti, A., and Sjödin, M. (2012). Full Code Generation from UML Models for Complex Embedded Systems. In *Second International Software Technology Exchange Workshop (STEW) 2012*.

Guizzardi, G., Herre, H., and Wagner, G. (2002). Towards Ontological Foundations for UML Conceptual Models. In Meersman, R. and Tari, Z., editors, *On the Move to Meaningful Internet Systems 2002: CoopIS, DOA, and ODBASE*, pages 1100–1117, Berlin, Heidelberg. Springer.

Hussein, M., Nouacer, R., and Radermacher, A. (2017). Safe adaptation of vehicle software systems. *Journal Microprocessors & Microsystems*, 52:272–286.

IBM (last visited 2019). *Instance specifications in UML.* https://www.ibm.com/support/knowledgecenter/SS8PJ7_9.5.0/com.ibm.xtools.modeler.doc/topics/cinstancespec.html.

Mammar, A. and Laleau, R. (2014). A Proved Approach for Building Correct Instances of UML Associations: Multiplicities Satisfaction. In *21st Asia-Pacific Software Engineering Conference*, volume 1, pages 438–445.

NoMagic (last visited 2019a). *Instance Specification*. https://docs.nomagic.com/display/MD183/Instance+Specification.

NoMagic (last visited 2019b). *Instance Table*. https://docs.nomagic.com/display/MD183/Instance+table.

OMG (2006). *Deployment and Configuration of Component Based Distributed Applications, v4.0*. OMG document formal/2006-04-02.

OMG (2017). *Unified Modeling Language (OMG UML), Version 2.5.1*. OMG Document formal/2017-12-05.

SCITEPRESS
SCIENCE AND TECHNOLOGY PUBLICATIONS