Towards a deep metamodelling based formalization of component models

Antonio Cicchetti
School of Innovation, Design and Engineering (IDT)
Mälardalen University, Västerås, Sweden
email: antonio.cicchetti@mdh.se

***Abstract*—Component-based software engineering (CBSE) is based on the fundamental concepts of components and bindings, i.e. units of decomposition and their interconnections. By adopting CBSE, a system is built-up by means of a set of re-usable parts. This entails that system's functionalities are appropriately identified so that implementing components can be accordingly selected. In turn, this means that each component-based design is at least made-up of two different instantiation levels, i) one for designing the system in terms of components and their interconnections, ii) and one for linking possible implementation alternatives for each of the existing components. In general, this twofold instantiation is managed at the same metamodelling level through the use of relationships. Despite such solutions are expressive enough to model a component-based system, they cannot represent the instantiation relationship between, e.g., a component and its implementations. As a consequence, validity checks have to be hard-coded in a tool, while the interconnection between component and implementation have to be managed by the user.**

In this paper we propose to exploit deep metamodelling techniques for implementing CBSE mechanisms. We revisit CBSE main concepts through this new vision by showing their counterparts in a deep metamodelling based environment. Interestingly, multiple instantiation levels enhance the expressive power of CBSE approaches, thus enabling a more precise system design.

***Index Terms*—model-driven engineering; component-based software engineering; component models; deep metamodeling; instantiation level;**

I. Introduction

The increasing complexity of contemporary software systems and the growing pressures to deliver products faster while still keeping high quality attributes demands appropriate development solutions. Component-based software engineering CBSE [1] is a well-established methodology that proposes to alleviate software development intricacy by studying the target application as an assembly of composable units (indeed, software components), each one addressing a particular aspect of the system. In this way, the complexity of the initial problem can be reduced through its partitioning into smaller subproblems. Moreover, time devoted to development and testing can be narrowed by promoting the reuse of already existing components across several software development projects [2].

Component-based system (CBS) specifications are intrinsically hierarchical: i) on the one hand, a component might be realised as the composition of several nested components; ii) on the other hand, a component might have multiple implementations distinguished by quality attributes, target platform, and so forth. Usually, modelling languages support such hierarchical structure in terms of relationships between a component and its sub-components, or between a component and its realisations, respectively. Despite this approach is powerful enough to represent complex CBSs from the expressiveness point-of-view, it requires a careful management of system validation. Notably, type correctness checking, that is verifying whether a component realisation is a valid instance of the component specification, has to be hard-coded in the tool. Moreover, this check should be re-executed each time changes were performed in the component specification and/or in its realisation. Besides, the relationship solution becomes quickly intricate with the growth of hierarchical decomposition levels. Practically, supporting more than two levels of component nesting poses relevant representation issues, as distinguishing the quality attributes of a parent component from the ones of its nested children.

Deep metamodelling [3] is a recent technique introduced in the model-driven engineering (MDE) research field to cope with multiple instantiation levels. It enhances the usual 4-layered metamodelling architecture [4] (also known as two-level metamodelling) by providing a recursive language extension/instantiation structure. In this respect, the deep metamodelling vision fits perfectly with CBSE methodology and its hierarchical decomposition of software systems [3]. In fact, deep metamodelling allows to represent a system and its components by means of arbitrary decomposition/instantiation levels.

This paper investigates the implementation of a component model by means of deep metamodelling mechanisms with the aim of verifying the feasibility of such a solution. The initial results illustrated in this work confirm the feasibility of the approach and meet the expectations of exploiting deep metamodelling mechanisms. Notably, hierarchical component structures can be represented in an easier way, while the conformance check of a component instance against a component specification is obtained by-construction. Despite both the component model and the deep metamodelling solution are specific, the discussion is kept generic enough to be reproducible with other component models and deep metamodelling approaches.

The paper is organised as follows: next section introduces CBSE together with a running example, which will be exploited in Section III to clarify the issues raising in considering

multiple instantiation levels. Section IV discusses the proposed formalisation of CBSE concepts through a deep metamodelling framework. Eventually, related works are discussed and conclusions are drawn in Sections VI and VII, respectively.

II. Instantiation relationships in CBSE

CBSE methodology relies on the notion of component, that is "*a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third party.*" [5]. Depending on the application domain, technological platform, and so on, the concept of component might include disparate characteristics, which are typically defined in a corresponding component model [6]. Therefore, a CBS is specified by adhering to a well-defined component model, that prescribes how components, their interconnections, and their deployment, look like.

For example, let us consider a simple Personal Navigation Assistant (PNA)[1] CBS as depicted in Figure 1: it includes `GPS Receiver`, `Power Management`, `Navigation System`, and `UI` components (represented as boxes with names). For the purpose of this paper, it is sufficient to know that the `Navigation System` retrieves geo-positioning information from a `GPS Receiver` and delivers navigation data to a user interface (`UI` component). These interconnections are represented by means of named relationships linking component ports. More precisely, a triangle shaped port represents an (provided) output of a certain component, while a square represents an (required) input. Therefore, `Navigation System` gets `Position` information from `GPS Receiver` and, after computing relevant `Navigation Data`, it delivers them to the `UI`.

[1]The example has been taken from [7] and readapted for the purpose of this paper.

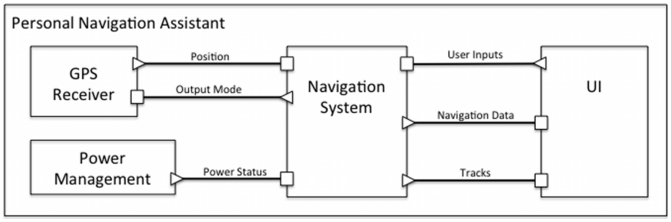

Fig. 1. A simple Personal Navigation System.

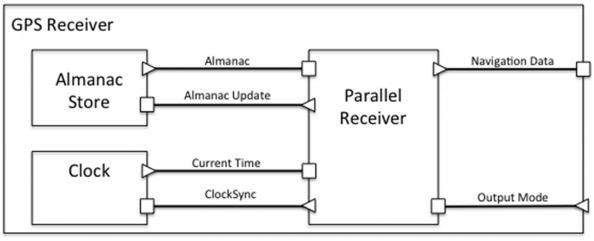

Fig. 2. A simple GPS receiver component.

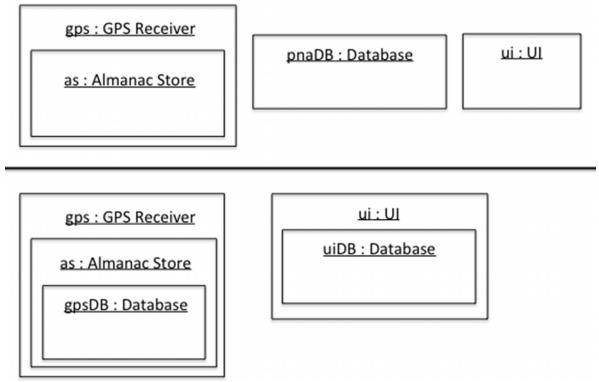

Fig. 3. Two (excerpts of) possible implementations for the PNA system.

In general, a component can include nested subcomponents, referred to as composite components [6]. This is the case of the `GPS Receiver`, which has a complex internal structure. As shown in Figure 2, GPS antennas (`Parallel Receiver`) have to coordinate their tasks with `Clock` and `Almanac Store`. In particular, satellite availabilities depend on the current time and are stored in an almanac.

Eventually, components are attached with one or more implementations, which can be distinguished by quality attributes, supported platforms, and so forth. Notably, for the PNA one might want to prioritise power consumption versus precision in a mobile phone while doing the opposite for a rescue device. Figure 3 illustrates two implementation alternatives for the PNA system: in the one shown on the top half of the picture, a single `Database` is shared between the implementations of `Almanac` and `UI` components, whereas the realisation shown on the bottom half exploits separate databases.

It becomes quickly evident that CBSE methodologies are intrinsically hierarchical: the generic notion of component assembly is instantiated by means of a specific component model (e.g., the simple one used in the example), which in turn is instantiated into a particular CBS (the PNA system). Even further, components can be realised in terms of other components and/or through implementations (as shown in Figure 2 and 3, respectively). In this respect, it is expectable that each CBS specification is made-up of only valid instances for the component model, the components defined in the system together with their implementations. Some of these instantiation relationships are managed by-construction: notably, a CBSE tool is built-up on a well-defined component model, hence the tool will support the design of CBSs by means of all and only the concepts offered by the selected component model (i.e. there is no need to verify that a CBS specification conforms to the component model).

A number of instantiation relationships however have to be checked case-by-case, and this validation step has to be addressed either by the designer, through appropriate constraints at modelling level (e.g. by means of OCL [8]), or hardcoded

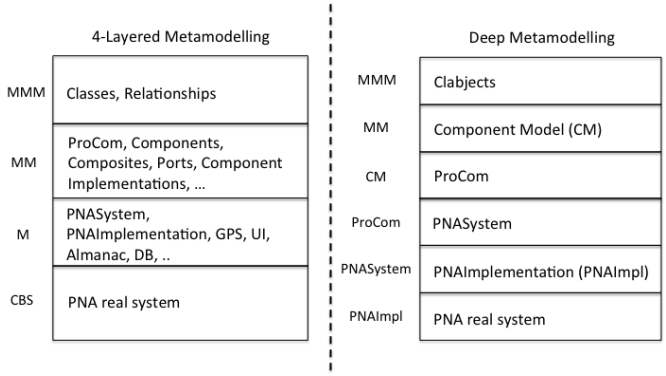

Fig. 4. A comparison between 4-layered and deep metamodelling architectures.

into the tool. For example, when specifying that the `GPS Receiver` composite component in Figure 1 is decomposed as in Figure 2, the tool should at least verify that input and output ports of the latter component specification are compatible with input and output ports of the former composite component (e.g., matching types). A similar reasoning has to be done when considering the interconnection between components and corresponding implementations. More specifically, every implementation of `GPS Receiver` should be compatible with every implementation of `Navigation System` when considering the exchange of `Position` and `Output Mode` data (e.g., the implemented setter and getter methods should match with their types). Regardless whether specified by the designer or if hardcoded in the tool, keeping consistent and up-to-date validity checks can be time-consuming and error-prone, especially when considering complex CBSs. Notably, if the system needed a more precise tracking of power status, the `Power Management` component could be refined as providing more details. In turn, these refinements should be propagated at implementation level by choosing appropriate component implementations for both `Power Management` and `Navigation System`.

III. On the need of a deep metamodelling solution

Current modelling techniques are usually based on a 4-layered metamodelling architecture [4]: a software system is represented by means of a model, that is an abstraction of reality for a given purpose. The model is created by following a set of well-formedness rules stated in a language definition, referred to as the metamodel. In other words, a metamodel defines the set of legal abstractions for a certain system. A model is said to conform to a metamodel if it adheres to the defined well-formedness rules. At the top of the 4-layered architecture there is the meta-metamodel, i.e. a unique minimal set of concepts needed to create all the possible languages. In this respect, the specification for the example introduced in Section II would be supported as shown on the left side of Figure 4: the `MMM` layer would be exploited to define a CBSE language based on a specific component model (at level `MM`), while the PNA system, its (sub-)components, bindings, and component implementations, would all be represented at the modelling level (i.e., `M`).

The conformance validity issues mentioned in Section II are due to the fact that a certain entity either pertains to the metamodel or to one of the models conforming to it. Moreover, at language level, realisation links defined between composite components and sub-components, and analogously between components and implementations, cannot guarantee conformance (i.e., they cannot impose type instantiation constraints). Technically, these relationships link concepts pertaining to different metamodelling layers that however cannot be represented in the typical 4-layered metamodelling architecture [3]. More specifically, the PNA system in Figure 1 is an instance of a certain component model, and at the same time the implementations in Figure 3 are instances of PNA components. In other words, a certain entity should play the role of a concept definition (`MM` level in Figure 4) and instance (`M` level in Figure 4) at the same time.

Multiple metamodelling layers allow to appropriately represent instantiation hierarchies, as depicted on the right side of Figure 4: a model can be equally considered as an instance conforming to the metamodel on the layer above and as a language definition (i.e. as a metamodel itself), for the layer below. In this way, it is be possible to define a component model as a metamodel a certain CBS specification conforms to, like it happens for `CM` and `ProCom` levels. In turn, the CBS specification would constitute a metamodel for which (sub-)component instances could be created (see `ProCom` and `PNASystem` levels, respectively). Eventually, implementations would be represented in a model conforming to a metamodel including simple component definitions (i.e., `PNAImpl`).

IV. A deep metamodelling formalisation for CBSE

This section illustrates the proposed formalisation of CBSE methodologies into a deep metamodelling framework. The formalisation proceeds step-by-step, from higher abstraction level concepts towards more and more concrete instantiations of them. In particular, we leverage a specific component model, namely ProCom [9], to implement the example presented in Section II. Moreover, we exploit MetaDepth [10] as support for concretising the formalisation proposal on a specific deep metamodelling environment. It is worth noting that, despite the component model and deep metamodelling solution are specific, the discussion is kept generic to be extensible to arbitrary component models and other deep metamodelling solutions.

In order to develop a system through CBSE methodologies, it is necessary to preliminarily adopt a specific component model [6]. In the most generic terms, a component model is made-up of components, bindings, and a platform. By adopting MetaDepth syntax, these concepts are specified as shown in Listing 1. In particular, `Component` nodes are bound by means of directional `Binding` edges (the direction is identified through attributes `bindingOut`, `bindingIn`, respec-

tively). A similar reasoning can be done for the `Deployment` relationship between `Component` and `Platform` nodes.

It is worth noting that, already at this stage it is possible to put modelling constraints: the `noSelfBinding` expression at line 18 prescribes that a component cannot be bound to itself. Moreover, `child` multiplicity at line 9 establishes that a `Composite` must have at list one nested component.

```
1 Model ComponentModel@*{
2  ext Node Component@*{
3   bindingIn: Component[*];
4   bindingOut: Component[*];
5   deployment: Platform[0..1];
6  }
7
8  ext Node Composite@*: Component{
9   child: Component[1..*];
10 }
11
12 ext Node Platform{
13  in: Component[*];
14 }
15
16 Edge Binding(Component.bindingOut,Component.bindingIn) {}
17 Edge Deployment(Component.deployment,Platform.in) {}
18 noSelfBinding@* : $Component.allInstances()->forAll(src,tgt
       | Binding(src.bindingOut,tgt.bindingIn) implies src!=
      tgt)$
19 }
```

Listing 1. Encoding of a generic component model.

The generic definition given in Listing 1 introduces the necessary CBSE concepts to create a specific component model. Notably, if we would like to define the ProCom component model, we would need to refine the generic bindings as ports, since ProCom adopts port-based interfaces. In particular, we introduce data ports and trigger ports, as illustrated in Listing 2, lines 4–7. Moreover, bindings have to be refined correspondingly (lines 18–19). It is important to notice that ProCom component model is defined in terms of, or better instantiates, the generic component model defined in Listing 1. This ensures, for instance, that `DataConnection` correctly binds a pair of `ProComComponents` through their `in_dataPort` and `out_dataPort`, respectively. Other alternatives, e.g. connecting a port with a child, would have raised type mismatch issues at validation time due to the type relationships defined before.

```
1 ComponentModel ProCom{
2  Component ProComComponent{
3   name: String {id};
4   in_dataPort: ProComComponent[*] {bindingIn};
5   out_dataPort: ProComComponent[*] {bindingOut};
6   in_triggerPort: ProComComponent[*] {bindingIn};
7   out_triggerPort: ProComComponent[*] {bindingOut};
8   parent: ProComComposite[0..1];
9  }
10
11 Composite ProComComposite: ProComComponent{
12  child: ProComComponent[1..*];
13 }
14
15 Platform ProComPlatform{
16 }
17
18 Binding DataConnection(ProComComponent.out_dataPort,
       ProComComponent.in_dataPort) { name: String {id}; }
19 Binding TriggerConnection(ProComComponent.out_triggerPort,
        ProComComponent.in_triggerPort) { name: String {id};
       }
20 Edge isChildOf(ProComComponent.parent, ProComComposite.
       child) {}
21 Deployment ProComDeployment(ProComComponent.deployment,
       ProComPlatform.in) {}
22 }
```

Listing 2. Encoding of (a subset of) the ProCom component model.

Once the component model has been defined, it is possible to model a CBS. In our case, we specify the PNA system introduced in Section II through ProCom, as shown in Listing 3[2]. In particular, the Navigation System, Power Management, and UI components in Figure 1 are modelled as `ProComComponents`, while the GPS receiver as a `ProComComposite` (see lines 2–13). Moreover, `DataConnections` are specified to bind the components appropriately, and implicitly define data ports for the corresponding components (lines 16–21).

Since GPS is defined as a composite, it is possible to define it as an assembly of sub-components. In this respect, Listing 3 shows the definition of `Almanac Store` at lines 23–25 according to the description of the GPS receiver depicted in Figure 2. Furthermore, by choosing the implementation alternative at the bottom of Figure 3, the almanac is defined as composite, thus allowing the introduction of a nested database component together with its quality attributes (lines 29–34). The nesting specification is completed with the definition of `isChildOf` relationships, as visualised at lines 36–38. Eventually, a platform is introduced to allow the deployment of the PNA system, and component deployments are specified accordingly (lines 41–47).

```
1 ProCom PNAModel{
2  ProComComposite GPS{
3   name = "GPS Receiver";
4  }
5
6  ProComComponent NS{
7   name = "Navigation System";
8  }
9  ProComComponent UI{
10  name = "UI";
11 }
12 ProComComponent PM{
13  name = "Power Management";
14 }
15
16 DataConnection(GPS.out_dataPort,NS.in_dataPort){name="
       Position";}
17 DataConnection(NS.out_dataPort,GPS.in_dataPort){name="
       OutputMode";}
18 DataConnection(PM.out_dataPort,NS.in_dataPort){name="
       PowerStatus";}
19 DataConnection(UI.out_dataPort,NS.in_dataPort){name="
       UserInputs";}
20 DataConnection(NS.out_dataPort,UI.in_dataPort){name="
       NavigationData";}
21 DataConnection(NS.out_dataPort,UI.in_dataPort){name="
       Tracks";}
22
23 ProComComposite AS{
24  name = "Almanac Store";
25 }
26
27    ...
28
29 ProComComponent DB{
30  name = "DB";
31  encryption: String = "NotDefined";
32  queryLanguage: String = SQL;
```

[2]Due to space limitations, some portions of the specification are omitted. The interested reader can download the full specification at http://www.es.mdh.se/~acicchetti/PNASystem.php .

```
33   WCET: int = 22;
34  }
35
36  isChildOf innerASDB(DB.parent,AS.child);
37  isChildOf innerAlmanac(AS.parent,GPS.child);
38  isChildOf innerUIDB(DB.parent,UI.child);
39  ...
40
41  ProComPlatform PNAPlatform{
42   name: String = "PNAPlatform";
43   CPU: String = "FPGA";
44   BUS: String = "EtherNet";
45  }
46
47  ProComDeployment GPSDeployment(GPS.deployment, PNAPlatform
        .in) {}
48  ...
49 }
```

Listing 3. Specification of the PNA system through ProCom.

An excerpt of the implementation of the PNA system is specified as shown in Listing 4. In particular, it illustrates the details for GPS, almanac, and database components (lines 2–18), together with the ones for UI and its nested database (lines 20–21), consistently to the implementation choice depicted at the bottom of Figure 3. Moreover, it shows the declaration of a platform and corresponding deployments at lines 33–34.

```
1 PNAModel pna{
2  GPS gpsImplementation{
3   name = "GPS1";
4  }
5
6  AS asImplementation{
7   name = "AS1";
8  }
9
10 DB dbImplementation1{
11  name = "DB1";
12  encryption = "none";
13  queryLanguage = "SQL";
14  WCET = 13;
15 }
16
17 innerDBAS(dbImplementation1,asImplementation);
18 innerAlmanac(asImplementation,gpsImplementation);
19
20 UI uiImplementation{
21  name = "UI1";
22 }
23
24 DB dbImplementation2{
25  name = "DB2";
26  encryption = "none";
27  queryLanguage = "SQL";
28  WCET = 22;
29 }
30
31 innerDBUI(dbImplementation2,uiImplementation);
32
33 PNAPlatform platform {}
34 GPSDeployment(gpsImplementation.deployment, platform.in);
35 ...
36 }
```

Listing 4. An excerpt of the specification of the PNA system implementation.

V. Discussion

At this point it is important to remark several relevant aspects related to the PNA system specification. From an instantiation procedure point-of-view, the deep metamodelling framework introduces correctness by-construction. Notably, once a system is defined as shown in Listing 3, it will be only possible to introduce component implementations as instances of the defined types (as in Listing 4). Even more important, the implementations have to obey the constraints set in the specification: `innerAlmanac` can only connect an implementation for the almanac with an implementation of a GPS (see line 18), while `GPSDeployment` can only be instantiated with an implementation for the GPS (see line 34). The check of such constraints comes "for free" by the system specification itself, which acts as a metamodel for the system implementation; on the contrary, the 4-layered metamodelling techniques would require additional coding and/or correctness rule definitions to check relationships consistency.

Another relevant aspect to notice is the ease of identification of type instances, which allows to set properties by component implementation, and link each of them to the appropriate component types. In particular, the two different implementations for the database are equipped with different quality attributes and can be included into different composites accordingly. Moreover, the deep metamodelling framework naturally supports the extension of attributes, making it possible to provide additional implementation details for component implementations (e.g. cost, size, and so forth) depending on target platform sensitiveness.

From a higher level of abstraction perspective, the deep metamodelling approach enables the definition of advanced modelling constraints. Notably, the component model might define modelling patterns/styles that later on will have to be preserved by system specifications in order to be successfully validated. This could include the number of components, the kind/number of allowed bindings, and so on. It is important to notice, once again, that similar constraints could be implemented also in the usual 4-layered metamodelling architectures. However, such a need would require implicit checks that in the long run can become time-consuming and error-prone.

As a drawback, the hierarchical arrangement of CBSs specification over multiple metamodelling levels could result as less intuitive and become less usable when dealing with complex systems. In this respect, it is very important to notice that the formalisation is intended to be transparent to the CBS designer, and should be considered as the underlying infrastructure over which a CBS tool would be implemented. MetaDepth is a text-based deep metamodelling environment, and as a consequence this work adopts the same approach. Nonetheless, other existing deep metamodelling tools have already demonstrated the implementability of diagrammatic layers over a base deep metamodelling technology (notably Melanee [11] and the DPF [12]).

The current description of this formalisation in inherently top-down, whereas an ideal CBSE approach would promote a bottom-up development, where useful pre-existing components are identified and picked-up from a repository [13]. In this respect, the component model can be still described as including a repository, and a valid CBS as being a collection of component repository elements. In this case, it would be up to the CBSE tool to create an appropriate instantiation hierarchy based on the selected components.

VI. Related Works

A preliminary choice in adopting a modelling language is deciding whether opting for a general purpose or a domain-specific language [14]. In general, the former solutions have embedded extension mechanisms (like the prototyping mechanisms for the UML [15]), while the latter demand proper language extensions through metamodelling activities. With respect to this paper, the former mechanisms provide more expressiveness through model instances (by inheritance), while the latter ones act on the metamodel to provide appropriate refinements. In both cases, the extensions are limited to the 4-layered metamodelling architecture that does not allow to introduce multiple instantiation levels.

The general need for better addressing multiple instantiation levels has been recognised in the last decade and corresponding solutions have been identified under the name of multilevel (or deep) modelling [10], [12], [16], [17]. In some cases, multilevel modelling techniques have been even used to implement domain-specific component-based systems, notably robots [18] and cloud services [19]. Nonetheless, to the best of our knowledge this is the first work that proposes a general formalisation of CBSE concepts, and in particular of component models, with the aim of enhancing current CBSE techniques.

The problem of managing multiple instantiation levels in CBSE has been already tackled by several works, as [7], [20], [21]. In general these works adopt a 4-layered metamodelling solution, that is, they typically exploit inheritance or other recursive relationships to provide support for containment/refinement modelling [7]. Therefore, they leave open the instantiation problems described throughout the paper.

VII. Conclusion and future works

This paper presents the first steps towards the formalisation of CBSE concepts in a deep metamodelling framework. Component-based systems have an intrinsic hierarchical structure and frequently exploit the "type-instance" pattern [3]. These characteristics have been identified as problematic to be implemented in the usual 4-layered metamodelling architecture and require better support. In this respect, the formalisation illustrated in this work shows promising improvements and gains with regard to both expressiveness and correctness checking.

Future investigation directions will include a more extensive experimentation of deep metamodelling techniques, especially focusing on the adoption of different component models, in order to verify the malleability of deep metamodelling in component adaptation/reconfiguration scenarios [13]. Moreover, the formalisation will have to be embedded in a CBSE tool to better evaluate the usability/scalability aspects related to both modelling and analysis tasks.

Acknowledgements

The author would like to thank Jan Carlson and Severine Sentilles for the interesting preliminary discussions around the topic covered in this paper.

References

[1] I. Crnkovic, "Component-Based Software Engineering for Embedded Systems," in *LMO*, 2006, p. 13.

[2] I. Crnkovic and M. Larsson, *Building Reliable Component-Based Software Systems*. Artech House, Inc., 2002.

[3] J. D. Lara, E. Guerra, and J. S. Cuadrado, "When and how to use multilevel modelling," *ACM Trans. Softw. Eng. Methodol.*, vol. 24, no. 2, pp. 12:1–12:46, Dec. 2014.

[4] J. Bézivin, "On the Unification Power of Models," *Software and System Modeling*, vol. 4, pp. 171–188, 2005.

[5] C. Szyperski, *Component Software: Beyond Object-Oriented Programming*, 2nd ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002.

[6] I. Crnkovic, S. Sentilles, V. Aneta, and M. R. V. Chaudron, "A classification framework for software component models," *IEEE Trans. Softw. Eng.*, vol. 37, no. 5, pp. 593–615, Sep. 2011.

[7] T. Lévêque and S. Sentilles, "Refining extra-functional property values in hierarchical component models," in *Proceedings of the 14th International ACM Sigsoft Symposium on Component Based Software Engineering*, ser. CBSE '11. New York, NY, USA: ACM, 2011, pp. 83–92.

[8] Object Management Group (OMG), http://www.omg.org/spec/OCL/2.0/PDF.

[9] S. Sentilles, A. Vulgarakis, T. Bureš, J. Carlson, and I. Crnković, "A Component Model for Control-Intensive Distributed Embedded Systems," in *Proceedings of CBSE*. Springer Berlin, 2008, pp. 310–317.

[10] J. de Lara and E. Guerra, "Deep meta-modelling with metadepth," in *Proceedings of the 48th International Conference on Objects, Models, Components, Patterns*, ser. TOOLS'10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 1–20.

[11] C. Atkinson and R. Gerbig, "Melanie: Multi-level modeling and ontology engineering environment," in *Proceedings of the 2Nd International Master Class on Model-Driven Engineering: Modeling Wizards*, ser. MW '12. New York, NY, USA: ACM, 2012, pp. 7:1–7:2.

[12] Y. Lamo, X. Wang, F. Mantz, W. MacCaull, and A. Rutle, "Dpf workbench: A diagrammatic multi-layer domain specific (meta-)modelling environment," in *Computer and Information Science 2012*, ser. Studies in Computational Intelligence, R. Lee, Ed. Springer Berlin Heidelberg, 2012, vol. 429, pp. 37–52.

[13] S. Becker, H. Koziolek, and R. Reussner, "The palladio component model for model-driven performance prediction," *J. Syst. Softw.*, vol. 82, no. 1, pp. 3–22, Jan. 2009.

[14] T. Kosar, N. Oliveira, M. Mernik, J. M. Pereira Varanda, M. Črepinšek, D. Da Cruz, and P. Henriques Rangel, "Comparing general-purpose and domain-specific languages: An empirical study," *Computer Science and Information Systems*, vol. 7, pp. 247–264, 2010.

[15] Object Management Group (OMG), "UML Superstructure Specification V2.3," http://www.omg.org/spec/UML/2.3/Superstructure/PDF/, 2011, [Online. Last access: 11/04/2012].

[16] C. Atkinson, M. Gutheil, and B. Kennel, "A flexible infrastructure for multilevel language engineering," *IEEE Trans. Softw. Eng.*, vol. 35, no. 6, pp. 742–755, Nov. 2009.

[17] B. Neumayr, K. Grün, and M. Schrefl, "Multi-level domain modeling with m-objects and m-relationships," in *Proceedings of the Sixth Asia-Pacific Conference on Conceptual Modeling - Volume 96*, ser. APCCM '09. Darlinghurst, Australia, Australia: Australian Computer Society, Inc., 2009, pp. 107–116.

[18] C. Atkinson, R. Gerbig, K. Markert, M. Zrianina, A. Egurnov, and F. Kajzar, "Towards a deep, domain specific modeling framework for robot applications," in *Proceedings of the First Workshop on Model-Driven Robot Software Engineering (MORSE)*. CEUR-WS, 2014. [Online]. Available: http://ceur-ws.org/Vol-1319/

[19] A. Rossini, J. de Lara, E. Guerra, and N. Nikolov, "A comparison of two-level and multi-level modelling for cloud-based applications," in *Modelling Foundations and Applications*, ser. Lecture Notes in Computer Science, G. Taentzer and F. Bordeleau, Eds. Springer International Publishing, 2015, vol. 9153, pp. 18–32.

[20] J. Odell, "Power types," *JOOP*, vol. 7, no. 2, pp. 8–12, 1994.

[21] R. C. Goldstein and V. C. Storey, "Materialization," *IEEE Trans. on Knowl. and Data Eng.*, vol. 6, no. 5, pp. 835–842, Oct. 1994.