Consistent Extra-Functional Properties Tagging for Component and Connector Models

Shahar Maoz[1], Jan Oliver Ringert[1], Bernhard Rumpe[2,3], and Michael von Wenckstern[2]
[1] School of Computer Science, Tel Aviv University, Israel http://www.cs.tau.ac.il
[2] Software Engineering, RWTH Aachen University, Germany http://www.se-rwth.de
[3] Fraunhofer FIT, Aachen, Germany http://www.fit.fraunhofer.de

***Abstract*—We present a model-driven approach for adding extra-functional properties to component and connector (C&C) models. The approach is based on a tagging mechanism that allows non-invasive extensions of existing languages and their models, here C&C models, with attributes for extra-functional properties. Importantly, our language extension provides means for integrated formal analyses of the consistency of tagged values. Consistency ranges from type-safety and units of quantitative measures to complex dependencies across component hierarchies as well as between component definitions and their instances. We provide a framework for defining and checking rich consistency rules of extra-functional property values based on selection, aggregation, and comparison operators. Our work allows for independent definition and organization of tagged properties to support reuse across models and development stages. The approach is implemented within the MontiCore framework for the C&C architecture description language MontiArc.**

I. Introduction

Component and connector (C&C) models are used in many application domains of software engineering, from cyber-physical and embedded systems to web services and enterprise applications. Their structure consists of components at different containment levels, their typed interaction points, and connectors between them [17]. In addition to expressing functional properties, also extra-functional properties (EFPs) play an important role in successful development of C&C models [10], [24]–[26]. Typical examples of EFPs include worst-case-execution-time, memory and power consumption, security properties, and traceability [1], [21], [23].

We are interested in consistent definition of EFPs for C&C models, commonly expressed in C&C architecture description languages (C&C ADLs) [17]. Previous works in this direction have extended the meta-models of languages or defined special language profiles for adding EFPs to ADLs [4], [26]. These extension mechanisms are typically invasive to the language definition and thus impede extensibility. One approach for extending modeling languages without modifying their meta-model are tagging languages [9]. **The first contribution of this paper is a tagging language for non-invasive extension of C&C models with EFPs.** This new language supports the tagging of C&C elements in component definitions as well as in their instances.

The consistency of EFP values is crucial throughout the life-cycle of a system. Consistency checks of EFPs have been investigated for various development steps of C&C models, e.g., property definition [26], refinement and subtyping [14], evolution [2], and deployment [25]. **The second contribution of this paper is a framework for defining rich consistency rules for tagged EFP values in the context of C&C models.** Our consistency rules are specific to an EFP and C&C element and consist of selection, aggregation, and comparison operators. Rules select relevant C&C model elements, aggregate their EFP values, and compare them to determine the consistency of the checked element's EFP value.

We implemented our ideas within MontiCore [13] for the C&C architecture description language MontiArc [11].

The next section presents our running example. Sect. III presents necessary background. The two main contributions, our tagging language and our consistency framework, are presented in Sect. IV and Sect. V. We present the implementation and a discussion in Sect. VI, related work in Sect. VII, and a conclusion in Sect. VIII.

II. Running Example

As running example we use an excerpt of a wind turbine example adapted from [25], [28], as shown in Fig. 1 and Lst. 1. The turbine controller (component type `TurbineCtrl`) contains two brake controllers (component type `BrakeCtrl`) to compute force on the turbine and merge the calculated result with brake commands from the main controller `MainCtrl`.

A team of engineers developing the controller deals with various EFPs including the traceability of component implementations to the requirements catalog and the power consumption of components. The component `BrakeCtrl` is a safety relevant component and thus requires traceability for safety audits. This extra-functional property is added to the component type definition. As a requirement the power consumption of `TurbineCtrl` is limited to $4W$.

The team created an implementation of component type `BrakeCtrl` that consumes power of $1W$ and is traceable. In an attempt to improve safety and provide different instances for subcomponents `brCoA` and `brCoB`, the team uses an existing implementation that consumes $2010mW$. The chief architect decides to update the power consumption defined for type `BrakeCtrl` to the new value of $2010mW$.

A team member is unsure whether these updates are enough and the C&C model is consistent with respect to its EFPs. She is right to doubt consistency: on the component type level the power consumption of `TurbineCtrl` ($4W$) is smaller than two times the power consumption of `BrakeCtrl`

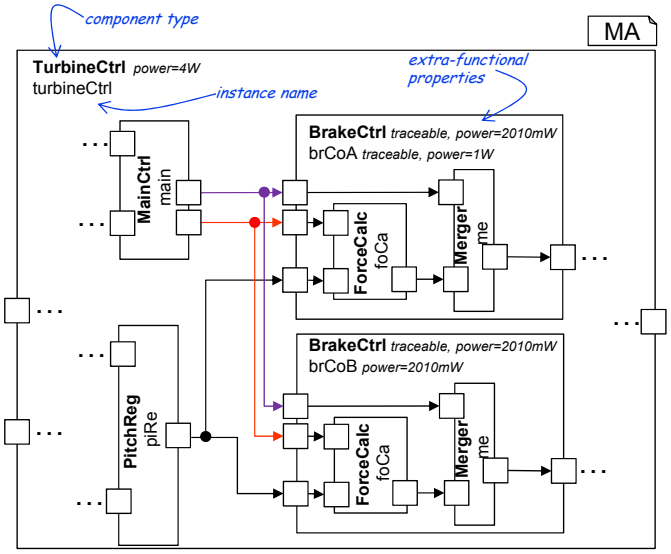

Fig. 1. Excerpt of slightly modified wind turbine example from [25], [28].

```
component TurbineCtrl {
  ports in AngularVelocity omega,
        in Velocity windSpeed,
        out Byte[] controlSignals;
  component MainCtrl main;
  component PitchReg piRe;
  component BrakeCtrl brCoA, brCoB;
  connect brCoA.brakeControl -> parkController.
      brakeControlA;
  ...
}
```

Listing 1. Definition of TurbineCtrl from Fig. 1 in MontiArc.

$(2010mW)$[1]. In addition, the component instance `brCoB` is not traceable despite what its type `BrakeCtrl` dictates.

Our solution allows the engineers to add the described properties to the C&C model and component type definitions and to check their consistency.

III. Preliminaries

We start off with background on C&C models, extra-functional properties, and tagging languages.

A. Component and Connector Models

Component and connector models describe components, their points of interaction, and their hierarchical composition. We repeat a definition of C&C models as, e.g., given in [15], in Def. 1, which represents the essence of component models [17] as formalized by ADLs ACME [7], AADL [5], and MontiArc [11], or used in the tools AutoFOCUS [12] and Simulink [29].

Definition 1 (Component and Connector model [15]): A C&C model is a structure $cncm = \langle Cmps, Ports, Cons, Types, subs, ports, type \rangle$ where

- $Cmps$ is a set of named components, $cmp \in Cmps$ has a set of ports $ports(cmp) \subseteq Ports$ and a (possibly empty) set of immediate subcomponents $subs(cmp) \subset Cmps$,
- $Ports$ is a disjoint union of input and output ports where each port $p \in Ports$ has a name, a type $type(p) \in Types$, and belongs to exactly one component $p \in ports(cmp)$,
- $Cons$ is a set of directed connectors $con \in Cons$, each of which connects two ports $con.src, con.tgt \in Ports$ of the same type, which belong to two sibling components or to a parent component and one of its immediate subcomponents, and
- $Types$ is a finite set of type names.

C&C models from Def. 1 are well-formed iff no component is its own (transitive) subcomponent and has at most one direct parent and subcomponents are connected legally (see [15] and [22] for complete definitions).

While some formalisms directly express C&C models, e.g., [12], [29], others provide component and connector type definitions and their instantiation to define C&C models, e.g., [5], [11]. We use excerpts of component type definitions of the ADL MontiArc in Def. 2.

Definition 2 (Component Type Definition): A component type definition is a structure $ct = \langle cType, CPorts, CSubs, CCons \rangle \in CTDefs$ where

- $cType$ uniquely identifies the component type,
- $CPorts$ is a set of input and output port definitions where each port $p \in CPorts$ has a name and a type,
- $CSubs \subset Name \times CTDefs$ is a set of named subcomponent declarations, and
- $CCons$ is a set of directed connector definitions $con \in CCons$, each of which connects two port definitions $con.src, con.tgt$ of the same type, which belong to two sibling subcomponent declarations or to a component type definition and one of its subcomponent declarations.

A component type $t \in CTDefs$ is instantiated to a C&C model by creating a component for c with subcomponents, that are instances of $t.CSubs$ with connectors according to $t.CCons$. For a more detailed definition including well-formedness rules and instantiation see [22].

B. Extra-Functional Properties

Extra-functional properties are attributes of a system or subsystem that do not directly describe functionality. Various definitions of extra-functional properties exist [1], [8], [21], [23], [30]. Typical examples of extra-functional properties include worst-case-execution-time, memory consumption, security properties, usability, maintenance, interoperability, testability, understandability, modifiability, reusability, correctness, flexibility and also legal, economic, or cultural constraints [1], [21], [23]. Extra-functional properties can be quantitative, e.g., memory consumption, or qualitative, e.g., a security level or a Boolean property indicating whether software is portable [20].

In this paper we focus on quantitative properties with clear mathematical semantics such as power consumption, traceability, or modes of encryption.

C. Tagging Language

Greifenberg et al. [9] presented a mechanism to derive tagging languages for domain specific modeling languages.

[1]Note, the described *instantiation* with ($1W$ for `brCoA`) could be consistent (depending on the other subcomponents), but the component *types* are not!

The mechanism uses two languages: the *tag model*, which decorates the domain specific model with additional information, and the *tag schema*, which defines the tag types that are used in the tag model.

Defining a tagging language based on this mechanism has the following advantages: (1) The model will be kept clean and, therefore, easy to read, not polluted with extra information; (2) Inherent separation of concerns [3], as different people can decorate the domain specific model with their own separated tagging models; and (3) The tagging language references the elements by their concrete syntax as they are defined in the domain specific model.

IV. C&C Tagging Language

This section presents our first contribution, namely, the C&C tagging language, which allows one to add extra-functional properties (encryption, ASIL level, traceability, memory usage, power consumption, latency, etc.) from different domains (security, safety, runtime, etc.) to existing C&C models, without changing them. Our tagging language enables tagging of all C&C elements from Def.s 1 and 2:

1) component definitions *CTDefs*, e.g., `BrakeCtrl`, and instances *Cmps*, e.g. `brCoA`;
2) port definitions *CPorts*, e.g., `BrakeCtrl.pitchBrake`, and port instances *Ports*; and
3) connector definitions *CCons*, e.g., `brCoA.brakeControl -> parkController.brakeContolA`, and connector instances *Cons*.

A. Tag Schema Definitions

The tag schema defines the types of the tags used to decorate C&C models. One may view tag schemes as meta-models.

Similar to Greifenberg et al. [9], we have the following tag types: (1) **simple tags**, when one only cares whether a C&C element is or is not tagged with this information, similar to Java's marker interface; (2) **valued tags**, decorating a C&C element with a tag containing a value, such as `Boolean`, `Number`, `String`, enumeration value or a JScience[2] quantity (e.g. `Power` or `DataAmount`); and (3) **complex tags** to store several values, such as estimated worst-case-execution time [26] `wcet = {time=800ms, confidence=50%}`.

Since complex tags consist of several simple or valued tags, the rest of this paper handles only the simple and value tags.

The `EmbeddedTagSchema` in Lst. 2 defines a tag schema for C&C models used in an embedded context. It contains one simple tag `traceable`, which can be applied for component instances and definitions, and three valued tags `power`, `encryption`, and `reliability`. All defined tags start with `tagtype`, have a name, and end with `for` plus the C&C element to which the tag type can be applied; the valued tag types have after the name additionally a colon followed by a data type. The data type of a valued tag can depend on the element decorated with it. This is the case for the `encryption` tag, where port definitions are tagged with a

[2]see http://jscience.org/api/javax/measure/quantity/Quantity.html

```
1 tagschema EmbeddedTagSchema {
2   tagtype traceable for ComponentInstance,
        ComponentDefinition;
3   tagtype power: Power for ComponentInstance,
        ComponentDefinition; // power consumption
4   tagtype encryption: [AES, RSA, DES, 3-DES]+
        for PortDefinition; // list of values
5   tagtype encryption: [AES, RSA, DES, 3-DES] for
         PortInstance; // one value
6   tagtype reliability: Number for
        ConnectorInstance;
7 }
```

Listing 2. Definition of example schema `EmbeddedTagSchema` for tagging C&C models and type definitions.

```
1 conforms to EmbeddedTagSchema;
2 tags Example1 {
3   tag BrakeCtrl with traceable;
4   tag TurbineCtrl with power = 4W;
5   tag BrakeCtrl with power = 2010 mW;
6   tag MainCtrl.pitchBrake with encryption = [AES
        , RSA];
7   tag BrakeCtrl.pitchBrake with encryption = [
        DES, 3-DES];
8 }
```

Listing 3. Tag model for tagging concrete EFPs values to component type definitions

```
1 conforms to EmbeddedTagSchema;
2 tags Example2 for turbineCtrl {
3   tag brCoA with traceable;
4   tag brCoA with power = 1 W;
5   tag brCoB with power = 2010 mW;
6   tag main.pitchBrake, brCoA.pitchBrake with
        encryption = AES;
7   tag brCoB.pitchBrake with DES;
8   tag brCoA.brakeControl -> parkController.
        brakeControlA with reliability = 0.8;
9 }
```

Listing 4. Tag model for tagging concrete EFPs values to component instances

set (+ sign in L. 4) containing one or more values of the enumeration defined inside square brackets; whereas, in contrast, tags for port instances contain exactly one enumeration value. This is due to the fact that a defined port can support multiple encryption modes, whereas a concrete port instance en-/decrypts its data using one concrete algorithm.

B. Tag Model Definitions

After the tag schema is defined, C&C elements can be tagged with it. Listings 3 and 4 tag C&C element definitions (`Example1`) and instances (`Example2`). Tag models have a header and body (e.g., Lst. 3, ll. 1-2 and ll. 3-8), containing additional information which is added to the C&C elements.

Every header starts with `conforms to` followed by the tag schema name to which the tag model definition conforms to. The header also includes the `tags` keyword and the tag model's name (e.g., `Example1`). Additionally, the header may contain an optional list of C&C elements (a comma separated list of names after the `for` keyword) to address these C&C elements' children directly in the body.

A body is a container for several tag definitions starting with a `tag` keyword and ending with a semicolon. Each tag definition has at least one C&C element name and one tag type name separated by the `with` keyword (e.g., Lst. 3, l. 3).

Valued tag types must additionally include the tag type's value, preceded by an equals sign (e.g., Lst. 3, l. 4). Multiple values are assigned by using a comma separated list inside square brackets (e.g., Lst. 3, l. 6).

Line 3 in Lst. 3 adds the `traceable` tag to the component instance `turbineCtrl.brCoA`, because the context is `turbineCtrl`[3] (l. 1). Line 8 in Lst. 4 shows that the domain expert decorating C&C elements needs no knowledge about the C&C meta-model and directly uses the concrete syntax of the C&C model (e.g., Lst. 1, l. 8).

V. Consistency of Extra-Functional-Properties

We now present the second contribution of our paper, namely, a framework for the definition of rich consistency rules for tagged extra-functional property values.

We distinguish between the consistency of EFP tags with their tag schema and the more interesting consistency of tagged EFP values in the context of the C&C model. Our rules for checking the consistency of tags and their schema are independent of the specific semantics of the respective EFP. In contrast, the consistency rules for values in the context of the C&C model are very specific to the expressed EFP.

A. Consistency of Tags and Tag Schema

The following rules check for consistency of tags and their tagging schema:

1) tag type names are unique per C&C model element kind
2) tagged C&C elements exist uniquely and are of the kind defined in the schema
3) every C&C element is tagged at most once per tag type
4) the tag value is of the data type defined in the schema
5) the unit of the tag is compatible with the unit in the schema, e.g., W and mW but not W and s
6) for complex tags the above applies to every value.

Note that rule 3 does not allow to tag a C&C element twice for the same EFP. While one could define strategies to resolve possible inconsistencies, e.g., considering maximal or minimal values, we added this rule to avoid inconsistencies.

B. Consistency of Tags and C&C Models

In addition to the consistency of tags with their tag schema, the consistency of a tagged EFP value may also depend on its context in the C&C model. More advanced examples of consistency relate to component instantiation and composition in C&C models. It is important to note that the consistency of a tagged EFP value may depend on multiple other C&C model elements and their relation. In addition, consistency may be very specific to the EFP type, e.g., allowing subsets of values or defining their bounds.

To address the challenge of ensuring consistency of tagged EFP values, we define a general framework based on consistency rules. First, each rule defines what tag of which kind of C&C model element it checks. Second, the rule specifies how to select relevant C&C model elements for the check. Third, the rule defines how to aggregate tagged values over the selected elements. Finally, the aggregated value is compared to the value of the checked element, to determine its consistency. We summarize the structure of consistency rules in Def. 3

Definition 3 (EFP Value Consistency Rule): A consistency rule is a structure consisting of:

checks name of tag and element checked by rule;
selection selects relevant C&C elements to check consistency;
aggregation aggregates values of selected elements; and
comparison compares values to decide consistency.

The next two subsections illustrate consistency definition rules according to Def. 3. Sect. V-B1 presents example rules for the consistency of EFP values in the context of component type instantiation. Sect. V-B2 presents example rules in the context of composition.

*1) **Instantiation Consistency Examples**:* Instantiation consistency checks whether the EFPs of C&C model instances conform to the EFPs of their type definitions. To simplify the definition of rules, we employ a general operator $typeOf : Cmps \rightarrow CTDefs$, which given a component instance returns its uniquely determined component type.

Rule 1 (InstTrace): If the component type definition is traceable, all instances have to be traceable:

- checks: tag `traceable` of $c \in Cmps$
- selection: $t := typeOf(c) \in CTDefs$
- aggregation: $v := t.\texttt{traceable}$
- comparison: $v \Rightarrow c.\texttt{traceable}$

In our example, component type `BrakeCtrl` is tagged as `traceable` in Lst. 3, l. 3. while component instance `brCoB` is not tagged as `traceable`. It will thus be reported by Rule 1 as inconsistent.

Rule 2 (InstPower): The power consumption of an instance is at most the power consumption of its type:

- checks: tag `power` of $c \in Cmps$
- selection: $t := typeOf(c) \in CTDefs$
- aggregation: $v := c.\texttt{power}$
- comparison: $v \leq t.\texttt{power}$

In our example, both component instances of type `BrakeCtrl` pass the check of Rule 2 with $1W \leq 2010mW$ for `brCoA` and $2010mW \leq 2010mW$ for `brCoB` (see Lst. 3, l. 5 and Lst. 4, ll. 4-5).

Rule 3 (InstEncryption): The encryption of a port instance must be in the encryption set of the port definition:

- checks: tag `encryption` of $p \in Ports$
- selection: $pt := THE[4]pt \in typeOf(p.parent[5]).CPorts : pt.name = p.name$
- aggregation: $v := pt.\texttt{encryption}$
- comparison: $p.\texttt{encryption} \in v$

In our example, the port instances `main.pitchBrake` and `brCoB.pitchBrake` pass Rule 1, while port instance `brCoA.pitchBrake` violates Rule 3: $\texttt{AES} \notin v = \{\texttt{DES}, \texttt{3-DES}\}$ (see Lst. 4, l. 6 and Lst. 3, l. 7).

[3] `turbineCtrl` is the top-level instance of the component type `TurbineCtrl` defined in Lst. 1

[4] Definite description operator $THEx : P(x)$ returns x satisfying $P(x)$.

[5] The parent of a port is the component it belongs to.

*2) **Composition Consistency Examples**:* Composition consistency checks whether the EFPs of C&C model elements are consistent across their composition. The following example rules address consistency on the type level. Similar rules can be defined on the instance level.

Rule 4 (CompPower): The combined power consumption of all subcomponents is at most the power consumption of the composed component:

- checks: tag `power` of $c \in CTDefs$
- selection: $S := c.CSubs$
- aggregation: $v := \sum_{(name,ct) \in S} ct.\texttt{power}$
- comparison: $v \leq c.\texttt{power}$

In our example, the component type `TurbineCtrl` contains subcomponents `brCoA` and `brCoB` of type `BrakeCtrl` and $v = 2010mW + 2010mW + \ldots \not\leq 4W$, i.e., the tagged value of $4W$ violates Rule 4 and is thus inconsistent.

Rule 5 (CompEncryption): A receiver port must support at least one encryption of its sender ports.

- checks: tag `encryption` of $p \in CPorts$
- selection: $P := \{p' \mid \exists con \in CCons : (p' = con.src \wedge p = con.tgt)\}$
- aggregation: $v := \bigcap_{p' \in P} p.\texttt{encryption}$
- comparison: $v \cap p.\texttt{encryption} \neq \emptyset$

In our example, the port `BrakeCtrl.pitchBrake` supports encryption `DES` and `3-DES` (Lst. 3, l. 7), and is a receiving port for `MainCtrl.pitchBrake` with encryption `AES` and `RSA` (Lst. 3, l. 6). Rule 5 will report port `BrakeCtrl.pitchBrake` as inconsistent.

The above examples of Rule 1 to Rule 5 show the expressiveness of consistency rules of our framework that cover various EFPs and C&C element relations.

VI. Implementation and Discussion

A. Implementation

We implemented the tagging language together with its EFP consistency checks in MontiArc [11], a textual C&C modeling framework. The workflow of processing MontiArc models is as follows: (1) The parser converts the textual input model to an abstract syntax tree (AST); (2) The AST is traversed to store all model definitions as symbols in the symbol table (ST); (3) Based on the AST and the model definition symbols, all C&C instances' symbols are created by (a) resolving the component extension chains, (b) binding all generics, and (c) recursively instantiating all subcomponent hierarchies.

Although the MontiArc tagging language is based on the same concepts and concrete syntax as the one presented in Greifenberg et al. [9], our approach adds tags to the ST while their method enriches the AST directly. Decorating the ST has the following advantages: (1) The ST represents the *semantic* model [6], [19], and, thus, in contrast to the AST (see `ArcConnector` and `ArcSimpleConnector` in [11]), each C&C element in Def. 1 and Def. 2 is represented by exactly one symbol; (2) Since MontiArc's AST is a mixture of C&C model definitions and instantiations for readability purposes, it is not possible to tag chains of instances (e.g., `turbineCtrl.brCoA.brakeControl`) differently as both of them are represented by the same AST node; the ST solves this problem by providing separate symbols for C&C definitions and for all C&C instances; and (3) Contrary to the AST, the ST is a graph with additional references making it possible to easily navigate through it and execute more complex selections as needed for consistency checks, e.g., in Rule 5.

B. Discussion

The ST's resolving mechanism – containing bidirectional navigation together with symbol filtering and adaption – as well as the ability to add several different EFP tags to the same C&C element symbol, allows to check constraints between different EFPs, such as time vs. data amount. We consider this to be a nice property of our work.

Regarding composition, the complex tag is a composition of tags (which can be complex themselves). This way, it is possible to logically group EFPs (e.g., power consumption and confidence) as is suggested by Shaw [27] and implemented by Sentilles et al. [26].

It is important to note that extra-functional properties may evolve, together with knowledge about a system [2], [14], [18]. They may also depend on the viewpoint of stakeholders [4], which can be solved by tagging the extra-functional tags again with the stakeholders name. Since it is possible to tag elements or tags several times, one tag can be tagged by different stakeholders. Since all the tags (including tags of tags) are stored in the same ST, consistency constraints can also be expressed between tags of tags with the same concepts shown in Sect. V. For simplicity, in this paper we did not discuss the tagging of one element multiple times. Rather than multiple tagging, we consider organizing ownership by using separate tag models. Meta-model approaches cannot easily do it but it is natural in our approach.

Note that depending on the EFP type and its composition and instantiation semantics, the consistency of composition on the type level and the consistency of instantiation, do not guarantee consistency of instance composition. Our presented port encryption semantics in Rule 3 and Rule 5 still allow for inconsistent instance compositions. We believe that a unified framework, as the one we have presented, is a first step towards reasoning about EFP consistency.

VII. Related Work

Espinoza et al. [4] annotate UML/MARTE models with quantitative EFPs. One of their main goals is to distinguish sources of EFPs, e.g., requirement vs. measurement during test. This could be done with complex tags containing the actual value and the value source.

Grunske [10] presents an evaluation framework for EFPs consisting of four elements: usage profile, evaluation model, composition algorithm, and evaluation algorithm. We focus on solutions for the composition and evaluation parts.

Sentilles et al. [26] present a meta-model for integrating non-functional properties into C&C models. Their model allows to specify multiple values per attribute with validity conditions, dependencies, and version information. Since our models are all text-based, external version control mechanisms such as Git or SVN can handle EFP's history. All the other information, e.g., validity condition or dependencies, can be expressed via complex tags. Finally, the complete information tagging is available in the symbol table for consistency checks.

Leveque et al. [14] present a way to express refinement of attribute values for instances and subtypes of components. Similar rules can be defined in our framework. For MontiArc, one can formulate EFP consistency constraints for refinements of port types (Java-like inheritance and generics) as well as for refinements of components (inheritance of component types).

Cicchetti et al. [2] introduce a framework for evolution of EFP values and present, as an example, how the change of the worst-case-execution time (WCET) of a component requires updating the WCET of its parent component. It is possible to extend our framework to support similar evolution scenarios.

Sapienza et al. [25] motivate the benefit of composing EFPs of components in embedded system design. They present a general classification of property composability and provide many examples. In their terminology, our rules for consistency of composed components compute non-emergent, directly composable properties. Additional types of composition identified by Sapienza et al. [25] require further information beyond the tagging language and C&C model.

VIII. Conclusion

We presented a mechanism to enrich existing C&C models with consistent extra-functional properties. The strengths of our approach are: (1) All EFPs are stored in separate files to avoid model pollution, (2) The tag schema is used to validate tag models to avoid tagging mistakes (typos, wrong units, wrong C&C element, etc.), (3) EFP-specific consistency rules between tagged C&C elements (C&C definitions as well as C&C instantiations) can be defined and verified.

We illustrated the two main contributions, our tagging language and consistency rule framework, using several examples. The examples cover many scenarios of consistency defined also in related work.

As future work we consider the application of extra-functional property tags to C&C views, with corresponding verification between views and models, extending [16].

Acknowledgements Part of this work was done while Shahar Maoz was on sabbatical as visiting scientist at MIT CSAIL. This research was supported by a Grant from the GIF, the German-Israeli Foundation for Scientific Research and Development.

References

[1] J. P. Cavano and J. A. McCall. A framework for the measurement of software quality. *SIGSOFT Softw. Eng. Notes*, 3(5):133–139, Jan. 1978.

[2] A. Cicchetti, F. Ciccozzi, T. Leveque, and S. Sentilles. Evolution management of extra-functional properties in component-based embedded systems. In *CBSE*, pages 93–102, 2011.

[3] E. W. Dijkstra. *A discipline of programming*, volume 1. prentice-hall Englewood Cliffs, 1976.

[4] H. Espinoza, H. Dubois, S. Gérard, J. L. M. Pasaje, D. C. Petriu, and C. M. Woodside. Annotating UML models with non-functional properties for quantitative analysis. In *MoDELS Int. Workshops*, pages 79–90, 2005.

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

[6] M. Fowler. *Domain-Specific Languages.* Addison-Wesley, 2010.

[7] D. Garlan, R. T. Monroe, and D. Wile. Acme: An architecture description interchange language. In *CASCON*, pages 169–183, 1997.

[8] M. Glinz. On non-functional requirements. In *RE*, pages 21–26, 2007.

[9] T. Greifenberg, M. Look, S. Roidl, and B. Rumpe. Engineering Tagging Languages for DSLs. In *MoDELS*, pages 34–43, 2015.

[10] L. Grunske. Early quality prediction of component-based systems - A generic framework. *Journal of Systems and Software*, 80(5):678–686, 2007.

[11] A. Haber, J. O. Ringert, and B. Rumpe. MontiArc - Architectural Modeling of Interactive Distributed and Cyber-Physical Systems. Technical Report AIB-2012-03, RWTH Aachen, february 2012.

[12] F. Huber, B. Schätz, A. Schmidt, and K. Spies. Autofocus: A tool for distributed systems specification. In *FTRTFT*, pages 467–470, 1996.

[13] H. Krahn, B. Rumpe, and S. Völkel. Monticore: a framework for compositional development of domain specific languages. In *International Journal on Software Tools for Technology Transfer (STTT)*, volume 12, pages 353 – 372, 2010.

[14] T. Leveque and S. Sentilles. Refining extra-functional property values in hierarchical component models. In *CBSE*, pages 83–92, 2011.

[15] S. Maoz, J. O. Ringert, and B. Rumpe. Synthesis of component and connector models from crosscutting structural views. In *FSE*, pages 444–454. ACM, 2013.

[16] S. Maoz, J. O. Ringert, and B. Rumpe. Verifying component and connector models against crosscutting structural views. In *ICSE*, pages 95–105. ACM, 2014.

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

[18] T. Mens, J. Magee, and B. Rumpe. Evolving software architecture descriptions of critical systems. *IEEE Computer*, 43(5):42–48, 2010.

[19] P. Mir Seyed Nazari, A. Roth, and B. Rumpe. An Extended Symbol Table Infrastructure to Manage the Composition of Output-Specific Generator Information. In *Modellierung 2016 Conference*, 2016.

[20] OMG. UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded Systems, 2011.

[21] A. Rawashdeh and B. Matalkah. A new software quality model for evaluating COTS components. *Journal of Computer Science*, 2(4):373–381, 2006.

[22] J. O. Ringert. *Analysis and Synthesis of Interactive Component and Connector Systems*. Aachener Informatik-Berichte, Software Engineering, Band 19. Shaker Verlag, 2014.

[23] G. C. Roman. A taxonomy of current issues in requirements engineering. *Computer*, 18(4):14–23, April 1985.

[24] M. Saadatmand, A. Cicchetti, and M. Sjödin. UML-based modeling of non-functional requirements in telecommunication systems. In *6th Int. Conf. on Software Engineering Advances (ICSEA)*, 2011.

[25] G. Sapienza, S. Sentilles, I. Crnkovic, and T. Seceleanu. Extra-functional properties composability for embedded systems partitioning. In *CBSE*, 2016.

[26] S. Sentilles, P. Stepan, J. Carlson, and I. Crnkovic. Integration of extra-functional properties in component models. In *CBSE*, 2009.

[27] M. Shaw. Truth vs. knowledge: the difference between what a component does and what we know it does. In *8th Int. Wrksp. on Software Specification and Design*, pages 181–185. IEEE, 1996.

[28] J. Suryadevara, G. Sapienza, C. C. Seceleanu, T. Seceleanu, S. E. Ellevseth, and P. Pettersson. Wind turbine system: An industrial case study in formal modeling and verification. In *FTSCS*, pages 229–245, 2013.

[29] MathWorks Simulink. http://www.mathworks.com/products/simulink/.

[30] S. Zschaler. Formal specification of non-functional properties of component-based software systems. *Software and System Modeling*, 9(2):161–201, 2010.