Increasing the Rigorousness of Measures Definition through a UML/OCL Model based on the Briand et al.'s Framework

Luis Reynoso[1], Marcelo Amaolo[1], Daniel Dolz[1], Claudio Vaucheret[1], Mabel Álvarez[2]
[1] University of Comahue
Buenos Aires 1400, Neuquén, Neuquén
{Luis.Reynoso, Marcelo.Amaolo, Daniel.Dolz,
Claudio.Vaucheret}@fai.uncoma.edu.ar
[2] Patagonia San Juan Bosco University, Argentina
Belgrano y Rawson (9100) Trelew, Chubut
mablop@speedy.com.ar

Abstract. The use of a formal definition of measures upon a metamodel assures that measures capture the software artifacts they intend for, improve repeatability and could facilitate the implementation of measures extraction tools. However, it does not assure that the measure captures the measurement concept it claims (like size, coupling, etc). For that purpose many formal frameworks had been defined. The well-known property-based framework proposed by Briand et al. defines the most important measurement concepts regardless the specific software artifacts to which these concepts are applied. In this article we define a UML/OCL model from the Briand's framework and we relate it with the formal definition of measures upon metamodels. We describe a set of well-formed properties that a measure should verify when capturing a measurement concept (which are derived from the model). We exemplify our approach through a thorough formal definition of UML statechart diagrams measures and its well-formed constraints of size measures.
Keywords: Measurement, Metamodeling, Property-based Framework, UML, MDA.

1 Introduction

Many authors argue that many difficulties may arise when measures are defined in an unclear or imprecise way. The lack of precision of what is defined by a measure may produce that the persons who builds the measure extraction tool, makes their own decision during implementation [29]. In this way, they can arrive at incorrect values of the measure. This situation arises when measures are not repeatable (the same result would not be produced each time a measure is repeatedly applied to a same artifact by a different person [14], [15]). Consequently, when measures are not repeatable, quality evaluators of models can take incorrect and undesirable decisions

of the external quality attributes of their models. So, a complete definition of measure should include not only in natural language but also in formal language, because how well a measure is understood will influence the way the measure is implemented and used. Given the relevance of model-driven engineering [1], [16] many authors make profit of the metamodel of software artifacts that are available to formally define a measure upon a metamodel. The metamodels give the more suitable framework of the main measured concepts and relationships upon which measures may be specified. The usage of the meta modeling approach for defining model-specific measures have been introduced in [3], [4], for defining class diagram measures upon the UML metamodel. Later, Reynoso et al. formally defined measures for OCL [17] upon the OCL metamodel [19], and for BPMN models [25]. In a model-driven development process companies need to measure their models [26] to enhance its quality [27], [28] and to address safety-critical concerns [26].

A precise definition of the measure will ensure repeatability, however their formal definition could neglect the mathematical properties of the measurement concept captured by the measure (the last is known as formal or theoretical validation). Briand et al. [24] argues about the importance of a precise definition of the mathematical properties that characterize the most important measurement concepts like size, complexity etc. regardless the specific software artifacts to which these concepts are applied. They provided a mathematical framework to define several concepts such as size, length, complexity, cohesion and coupling. The rigorous of the framework is provided by the mathematical underpinning. We were interested in defining a set of OCL properties derived of that framework that a formal definition of measures should verify according to each measurement concepts. So, in this paper we show a model to define a set of constraints from Property-based Framework of Briand el al. (PbFB) and then we exemplify how to use it in the formal definition of a statechart measure. We show how the formal definition of UML Statechart Diagram (SD) measures using a meta-modeling approach are defined and can be added with well-formed properties of the PbFB model. We use UML models because they become the primary artifacts, focus and products [23], [22] in recent Model-Driven Engineering (MDE) initiatives.

The advantages of a formal definition of a measure upon a metamodel are many: (1) the formal definition of measures: misunderstanding between the authors and the readers of the measures is avoided; (2) measures extraction tools can obtain the same results of a same measure; (3) experimentation is not hampered [2] due to the fact that an experiment and each of its replicas use measures which are repeatable; (4) measure value can be computed before and after a modification of design (or model transformation) is introduced, so measure can help to assess the quality of the diagram [20]; (5) they could facilitate the implementation of measures extraction tools, even more, they are used to generate a fully fledged measurement software [26].

Beside the aforementioned advantages of a formal definition of measure, a theoretical definition of measure constitutes a necessary constraint to assure that the measure captures a measurement concept [29]. In the same way the theoretical definition of a measure is a necessary conditions but not sufficient, due to the fact the measures should suffer an empirical validation, we argue that a formal definition of measure upon a metamodel is a necessary condition of its definition and it should be complemented with the add-on of well-formed properties derived of their theoretical definition.

Montperrus et al argues in [26] that measurement software can be generated from an abstract and declarative specification of metrics, from a model of metric specifications. We believe that the PbFB model we provide in this article will be also useful for measurement software, it helps to know and to monitoring how a measurement concepts are captured by a measure during the model-driven process. We believe that the model will be also helpful to academic purpose.

This paper starts in the next section defining the UML/OCL model for the property-based framework of Briand (PbFB) and shows the OCL properties for size. Section 3 gives a brief presentation of the measures for UML Statechart Diagrams defined by Lemus [10]. The statechart diagrams are the software artifacts upon which we will exemplify a set of formal definitions of measures and the application of the PbFB model. In addition, section 3 presents the UML Statechart metamodel and an example of an instance of the metamodel illustrated how measures can be captured. Section 4 provides the formal specification of the measures and the application of the PbFB Model. Finally, the last section presents some concluding remarks and future work. Appendix A shows the complete definition of properties of the PbFB model.

2 The PbFB Model

In the property-based framework of Briand [24] a system is represented as a class having a set of elements and a set of binary relationship between the elements of the system. In Figure 1 we depicted the System class and two related classes, Element and Relation classes. The system verifies the property that *all the relationships link elements of the system:*

```
context System
inv: self.getRelationships()->forall( r:Relation |
self.getElements()->includes(r.source) and
self.getElements()->includes(r.target) )
```

Given a system, *a system is considered a module if and only if its elements (or relationships) are a subset of the elements (or relationships) of the system which contains it.* We model a module as a system according to its definition. The *module* attribute in System class (Boolean type) identifies whether a system is a module[2]. The modules contained in a system are modeled using a relationship which links System with itself (see Figure 1). So, a module satisfies the following property:

```
context System
inv: if module then
           self.getElements()->forall(e: Element |
           self.system.getElements()->includes(e)) and
  self. getRelationships()->forall(r: Relation |
  system.getRelationships()->includes(r))
```

[2] A module can also be modelled as subclass of System but practically there is no difference between a system and a system module which can be distinguished with this attribute.

```
  else true
```

The elements of a module are connected to the elements of the rest of the systems by incoming and outgoing relationships. The *InputR()* operation obtains the set of relationships from elements outside module *m* to those of module *m*. *InputR()* is defined as:

```
context System:: inputR(): Set(Relation)
inv: if module then
self.system.getRelationships()->collect(r |
self.getElements()->includes(r.target) and not
self.getElements()->includes(r.source))
```

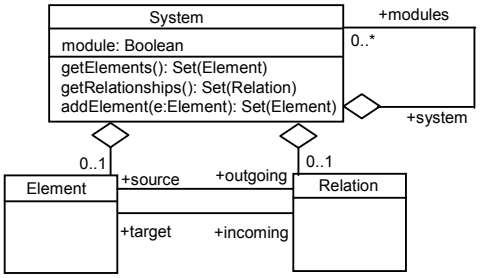

Fig. 1. A model used in the formalization of the PbFB

Similarly, it is possible to define *OutputR(),* a set of relationships from elements of a module *m* to those of the rest of the system.

Included defines an observer operations to verify that a system includes the relations and elements of another:

```
context System:: included(m:System):Boolean
inv: if self.module and m.module then
m.getElements()->forall(e |
    self.getElements()->includes(e)) and
m.getRelationships()->forall(r |
  self.getRelationships()->includes(r))
```

2.1 **Properties of Size.** We suppose that *size_md()* defines a measure which captures the size of the system. Size_md should also verify the following well-formed properties according to Briand et al.:

Non-negativity. The size of a system is non-negative.

```
context System:: size_md()
post non-negativity: result > 0
```

Null Value. The size of a system is null whether the system has no elements.

```
context System:: size_md()
inv null_value: self.getElements()->isEmpty()
  implies self.size_md() = 0
```

Module Additivity. The size of a system is equal to the size of its disjoint modules

```
context System:: size_md()
inv module_additivity:
self.modules->forall(m1,m2 |
  m1.getElements()->intersect(
  m2.getElements())->isEmpty() )
and
self.modules->collect(m1.getElements())->flatten() =
self.getElements()   implies
            self.size_md()= self.modules->collect(m |
            m.size_md())->sum()
```

The last property is equivalent to Size.IV property of Briand et al. Size.III is a particular case of Size.IV. Therefore, the following property is also valid:

Size Monotonocity Property. Adding elements to a system cannot decrease its size.

```
context System:: add_element(e: Element)
post monotonocity_property:
  self@pre.size_md() <= self.size_md()
```

An additional property is defined in Briand et al. which follows from the above properties. The size of a system is not greater than the sum of the sizes of its modules due to the presence of common elements:

```
context System:: merging_modules()
inv merging_modules:
if self.modules and self.getElements() = self.modules-
>collect(m: Module | m.getElements())->flatten() implies
self.size_md()= self.modules->collect(m | m.size_md())-
>sum()
```

Appendix A shows the formal definition of properties for cohesion and coupling according to the Briand et al. framework [24]. Properties for complexity, length are available in a technical report (see appendix A).

3 Measures for UML SD

The UML has now become the de facto standard software systems modeling and the UML State Diagram (SD) has become an important technique for describing the

dynamic aspects of a software system [12]. An SD contributes to the behavioural specification of a type in a model. A thoroughly definition of a set of measures for structural properties of UML SD is presented in [10] based on the hypothesis that structural properties of an UML SD (the software artifacts measured) have an impact on the cognitive complexity of modelers (subjects), and high cognitive complexity leads the UML SD to exhibit undesirable external qualities on the final software product [14], such as less understandability or a reduced maintainability [5], [6]. These measures are supposed to be good indicators of the understandability of such diagrams. This fact was empirically validated in [8], [9], [11]. These measures were defined following a method consisting of three main steps [7]: measure definition, theoretical validation and empirical validation. But initially the measures were informally defined using natural language. So, one contribution of this paper is to thoughtfully show its formal definition whose purpose was briefly described in a short paper [30].

3.1 The UML Statechart Metamodel

A SD describes possible sequences of states and actions through which the element instances can proceed during its lifetime as a result of reacting to discrete events (for example, signals or operation invocations). The abstract syntax for state machines is expressed graphically in UML StateChart Metamodel [18], which covers all the basic concepts of state machine graphs such as states, transitions, guards, etc. In this section we will first give an overview of the Metamodel explaining its main metaclasses and relationships and then we will show the whole metamodel. Secondly, we will describe an SD as an instance of the UML Statechart metamodel.

3.2 Overview of the Main Metaclasses

Every state machine has a top state (see Figure 2), usually a composite state, that contains all the other elements of the entire state machine. The graphical rendering of this top state within an SD is optional.

The State hierarchy has a State superclass and three subclasses, CompositeState, SimpleState and FinalState. This hierarchy, in fact, is part of the StateVertex hierarchy as it is shown in Figure 3, which also includes the PseudoState, SynchState and SubState classes. The composite state may contain any state of the StateVertex hierarchy.All the classes, attributes of classes and relationships previously depicted in Figures 2 and 3 are part of the UML Statechart metamodel depicted in Figure 4 [18].

Each State in an SD may have associated actions, such as entry, exit or a do-activity actions (see in Figure 4 the relationships between the State and Actions classes having the *entry*, *exit* and *doActivity* rolenames). Nevertheless, no more than one action of a specific type is allowed for a particular state.

Transitions usually connect two states, for example two Simple States, a Simple State with a Final State, etc. These connections are described in Figure 4 through two relationships between the StateVertex and Transition classes, where each of them identifies the source and target StateVertex which is connected through the transition.

So, any transition connects exactly a source to a target statevertex. From a StateVertex point of view, any state can have many incoming and outgoing transitions [13].

Within an SD, transitions may also be labeled with Guard and Events. This situation is modelled through the Guard and Event classes which are related to the Transition class (see the relationship between Transition and Guard classes with the *guard* rolename, and the relationship between Transition and Event classes with the *trigger* rolename).

The set of all transitions within an SD is modeled through a relationship between the StateMachine and the Transition classes.

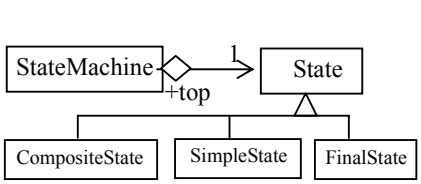

Fig 2. The StateVertex Hierarchy

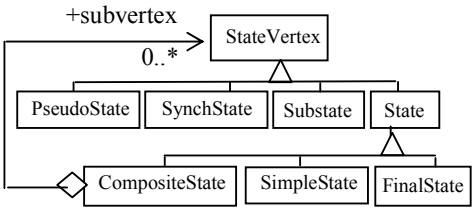

Fig 3. The top State of a StateMachine

3.3 A Metamodel Instance Sample

In order to understand both the UML Statechart metamodel and the specification of measures for UML SD we will give in this section, a sample of how an SD is represented as a metamodel instance. The diagram we will use is shown in Figure 5 and its representation as an instance metamodel is depicted in Figure 6.

Although the SD example is rather simple because it is composed of five simple states and two composite states, its representation as a metamodel instance is quite complicated and involves more than thirty objects.

The SD shown in Figure 5 describes a simple quality control process for testing incoming raw materials. At the beginning the received material is in the NoVerification state. The raw material is tested in house (Testing state) when the supplier of the material is not a certified supplier otherwise the raw material state is changed to the Accepted state. This is modelled through two labels, check[CertifiedSupplier] and check[NoCertifiedSupplier] which represent an Event ('check') having each of them a Guard ('CertifiedSupplier' or 'NoCertifiedSupplier'). When a raw material is in the Testing state the *entry* activity of the state is triggered, and the material is tested. The test has two possible results: approved or rejected. According to each situation the raw material will change to the Accepted or Rejected state respectively. A report is filled when materials are rejected. Accepted raw materials are stored within the warehouse (and the stock is updated) whereas rejected raw materials are returned to the sender. In both cases, the process of reception for the raw material ends.

Now we will explain the diagram shown in Figure 5 as an instance of the UML Statechart Metamodel. Figure 6, an object diagram, consists of many objects which are instances of the UML statechart metamodel' metaclasses (explained in section 3.1 and 3.2). In order to easily refer to the objects of the metamodel object diagram of Figure 6 we have named a vertical and an horizontal axes (of the SD) with letters and numbers in order to use them as a simple cartesian coordinate system. For example,

in the position A1, a Statemachine' object, named RawMaterialReception, is shown to represent the diagram itself. Because every state machine has a top state that contains all the other elements of the entire state machine, the RawMaterialReception' Statemachine object has a top state, a composite state, depicted in the A2 position of the diagram. This composite state, which has no name, composes four states: an initial state (a PseudoState object having the 'initial' value in the kind attribute [18], see position C1 of Figure 6), two simple states (named NoVerification and WithinWarehouse, see positions F1 and D2 of Figure 6 respectively), a final state (position B2 of Figure 6) and a composite State (position A4 of Figure 6). This last composite state has three simple states: Accepted (position C4), Testing (position G4) and Rejected (position I5) states.

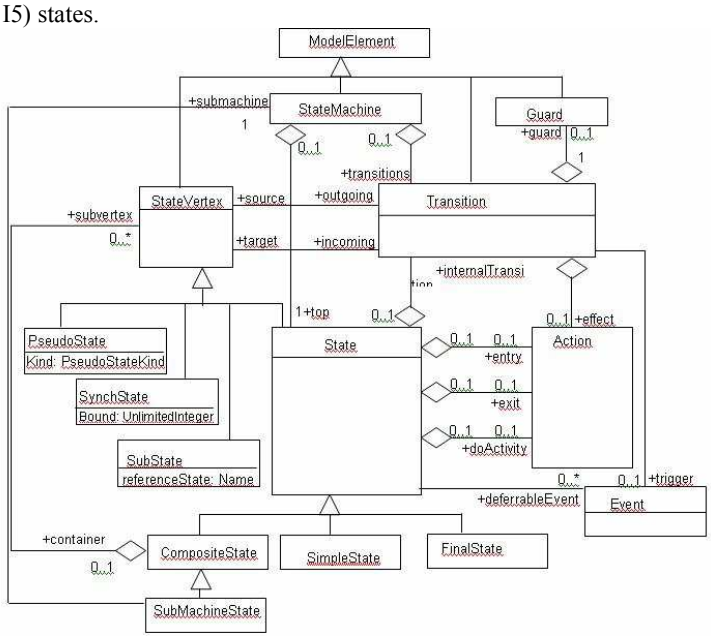

Fig 4. The SD Metamodel

Figure 6 also shows eight transitions between states. The transition connects different kinds of states. For example:

1. Transitions between Simple states: a transition between the NoVerification and Accepted states is shown in the E3 position. Other transition between simple states are depicted in C3, D4, G3, H4 positions.
2. Transitions between Simple and Final States: a transition between the WithinWarehouse simple state and the Final state is shown in C2 position. Another transition between a simple and a final state is shown in the I1 position.
3. Transition between Initial and Simple State: a transition between the initial and the NoVerification states is shown in the E1 position.

Each of the aforementioned transitions connects a source state to a target state. The relationships between the StateMachine object and the transition objects are not shown in Figure 6 to avoid clutter the readability of the diagram.

Four transitions (shown in I1, H4, D4 and C3 positions) have associated a Guard object (shown in the I2, H5, D5 and B3 positions respectively).Similarly, two transitions (see E3 and G3 positions) have an Event (D3 and G2 positions) with a Guard associated (F3 and H3 positions) to the event.

3.4 Specification of Measures

In this section we will show the specification of SD measures using the UML Statechart metamodel [18].

The specification of the measures relies on three query operations:

1. Alltransitions operation, defined in the StateMachine metaclass, obtains the set of transitions in an SD.
2. AllStates operation, defined in the StateMachine metaclass, selects the set of all the states within an SD.
3. AllSubStates operation, used by the two previous operations and defined in the StateVertex metaclass, obtains the set of all Subvertex included in a SD. It is recursively defined.

Their OCL definitions are shown below:

```
context StateMachine::allTransitions::Set(Transition)
body: result =  self.transitions>
union(self.allSubStates().internaltransitions)
context StateMachine::allStates:Set(State)
body : result = self.top.allSubstates()

context StateVertex::allSubstates::Set(StateVertex)
body: result =if self.oclIsKindOf(CompositeState)
then self.oclAsType(CompositeState).subvertex->union
self.oclAsType(CompositeState).subvertex
-> select (s:StateVertex| s.allSubstates())
else Set{} endif
```

For obtaining the value of each SD measure of Lemus [10] we defined in the StateMachine metaclass an operation with the same name as the measure. So, 14 operations were defined. Using the AllSubState operation we can define the value of many SD measures. This operation returns the set of all the states (of different kinds: initial, final, simples, etc.) included in a diagram, even those states which are part of composite states. For example, selecting from the allsubstate operation result, those states of an SD which have associated a doActivity action, it is possible to obtain the Number of Activity (NA) measure's value. The quantity of objects selected represents the value of the NA measure.

```
context StateMachine::NA():Integer
body: result = self.top.allSubstates()->select(s |
s.oclType(State)  and s.doActivity->notEmpty() )-> size()
```

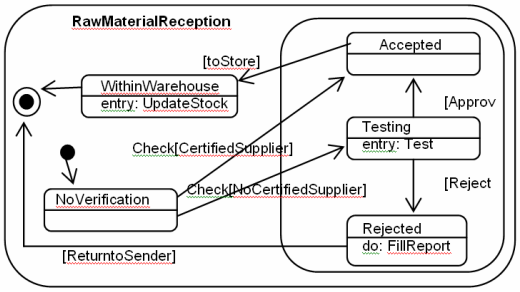

Fig 5. A state machine called RawMaterial Reception

When the NA() operation is requested in the rawmaterialreception object of Figure 6, the obtained result is 1, due to the fact that only one state (see I5 position) has a *doActivity* action associated. In a similar way we can obtain the value of Number of Simple State (NSS) measure of Lemus [10]. First, we select those StateVertex which are instance of SimpleState class, then we obtain the quantity of the object contained in this selection. When the NSS() operation is requested in the rawmaterialreception object of Figure 6, the result is 5 (the five simple states). The rest of the measures of [10] are formally defined in a technical report [31].

```
context StateMachine::NSS():Integer
body: result = self.top.allSubstates()->select (s |
s.oclType(SimpleState))-> size()
```

4 Example of NSS and its properties

Lemus et al. [10] using the property-based framework of Briand et al. show that NSS, NA, NLCS, NT, NCS, NCT, NCTG1 are size measures. In their theoretical validation (see [10]) a UML Statechart Diagram is considered a system (i.e. is the StateMachine class in Figure 4) the elements are states (i.e. is the stateVertex class in Figure 4), the relations are transitions (the Transition class in Figure 4) and modules are considered a subset of states and transitions of the SD.

In our formal definition, the StateMachine class is added with a self-relationship to model the modules of a system. The getElements() and getRelationships are defined in the following way:

```
context StateMachine::getElements()::Set(State)
body: self.allStates()
context StateMachine::getRelationships():Set(Relation)
body : self.alltransitions()
```

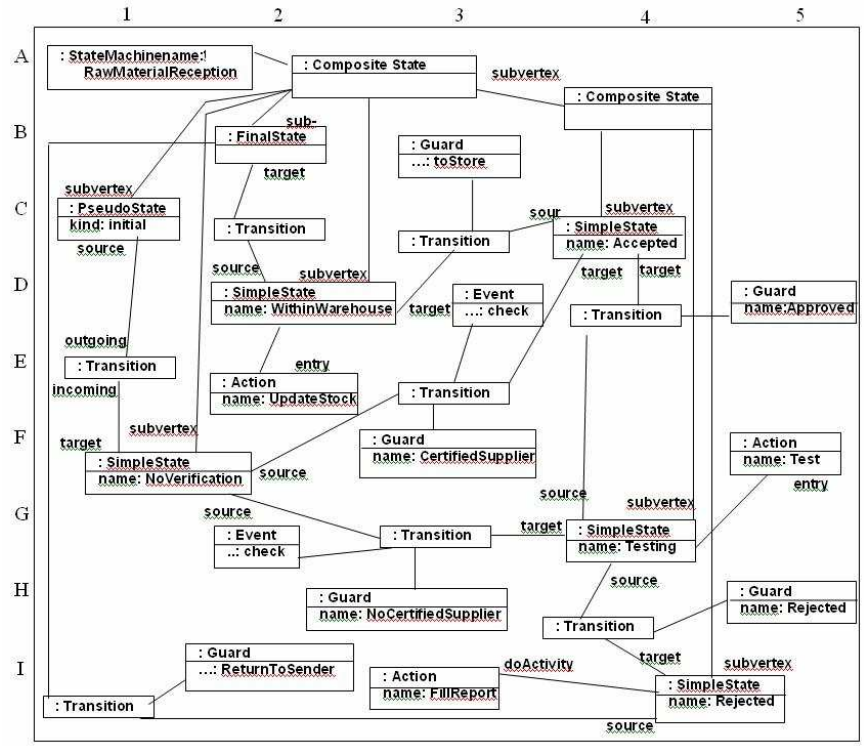

Fig 6. An object diagram (metamodel instantiation) for the statechart of Figure 5

So we can add a set of well-formed properties to each size measure definition according to the properties of size defined in section 2.1. For example for the NSS definition we add the following properties:

```
context StateMachine::NSS():Integer
body: result = self.top.allSubstates()->select(s |
s.oclType(State)  and s.doActivity->notEmpty() )-> size()
post non-negativity: result > 0
inv null_value: self.getElements()->isEmpty() implies
self.NSS() = 0

inv module_additivity: self.modules->forall(m1,m2 |
m1.getElements()->intersect( m2.getElements())->isEmpty()
)
and
self.modules->collect(m1.getElements())->flatten() =
self.getElements() implies
self.NA()= self.modules->collect(m|m.NSS())->sum()
```

A complete list of properties for the rest of measurement concepts is included in Appendix A.

5 CONCLUSIONS

The quality of models is acquiring more relevance within the introduction of model-driven engineering. In order to assess the quality of a model, it is unavoidable the definition and application of measures for them. Many measures had been proliferated in literature but they are not correctly defined, and the worst, they are not integrated with the software artifact and with the measurement concept to which they are related (or they claim to be). Measures defined upon a metamodel not only had benefits in terms of the final product but also are ready to support ongoing model-driven development process. Their definition can be derived from model to model through model transformation. The main contribution of this paper is the definition of a UML/OCL model which captures the property-based framework of Briand et al [24]. The model includes the formal properties defined by Briand et al. for size, cohesion, length, complexity and coupling measure defining a set of OCL constraints upon a small UML model. In this paper we used the statechart measures of Lemus et al. [10], [11] to show how to connect the formal definition of measures upon the UML statechart metamodel and the OCL properties the measure should verify to capture the measure concept of size defined by Briand et al.[24].

Refactoring techniques [21] and MDA-based system for measures extraction which improve the design of UML systems can take advantage of using the formal de definition of measures and the constraints related to the measurement concepts their capture. Measure value can be computed before and after the refactoring is applied, to evaluate the change according the quality of the statecharts [12], constraints should be valid as part of assert-restrictions of the system. Using this approach, the rigorous definition of measure can be useful to build solid software measurement tools and to verify that measure constraints are verified by its metamodel, models and objects.

6 ACKNOWLEDGMENTS

This research is part of the 048/12 'Hacia el Fortalecimiento de la Sociedad en el Uso y Aplicación Geoespacial y las TICS' of Patagonia San Juan Bosco University, Chubut (Argentina) and the 'Modelos y Tecnologías para Gobierno Electrónico' of Comahue University, Neuquén (Argentina)".

7 References

1. Atkinson, C., Kühne, T.: Model-Driven Development. A Metamodeling Foundation, IEEE Software, 20(5), 2003, pp. 36- 41.
2. Baroni, A. L.: Formal Definition of Object-Oriented Design Metrics. Master of Science in Computer Science Thesis, Vrije Universiteit Brussel, Belgium, 2002.

3. Baroni, A. L. , Braz, S., Brito e Abreu, F.. Using OCL to Formalize Object-Oriented Design Metrics Definitions. In Proceedings of QUAOOSE'2002, Malaga, Spain, 2002.
4. Baroni, A. L., Brito e Abreu, F.: A Formal Library for Aiding Metrics Extraction. In Proceedings of the Int. Workshop on Object-Oriented ReEngineering at ECOOP'2003. 2003.
5. Briand, L. C., Bunse, L. C., Daly, J. W. A Controlled Experiment for evaluating Quality Guidelines on the Maintainability of Object-Oriented Designs. IEEE Trans. on Softw. Eng., Vol. 27 Nº 6, 2001, pp. 513-530.
6. Briand, L. C., Wust, J. , Ikonomovski S. and Lounis, H.: Investigating Quality Factors in Object Oriented Designs: An Industrial Case-Study. 21st Int. Conf. on Software Engineering, 1999, pp. 345-354.
7. Calero, C., Piattini, M., Genero, M. Method for Obtaining Correct Metrics. In Proceedings of the 3rd Int. Conference on Enterprise and Information Systems (ICEIS`2001), 2001, pp. 779-784.
8. Cruz-Lemus, J.A., Genero, M., Manso, M.E. and Piattini, M. Evaluating the Effect of Composite States on the Understandability of UML Statechart Diagrams. In Proceedings of MoDELS 2005, LNCS 3713, 113-125, 2005.
9. Cruz-Lemus, J. A., Genero, M., Morasca, S., and Piattini, M. (2007). Assessing the Understandability of UML Statechart Diagrams with Composite States - A Family of Empirical Studies (submitted to Empirical Software Engineering).
10. Cruz-Lemus, J. A., Genero, M., Piattini, M. Metrics for UML Statechart Diagrams. In Proceedings of Metrics for Conceptual Models. Imperial College Press, UK. 2005.
11. Cruz-Lemus, J. A., Genero, M., Piattini, M. and Toval, A. An Empirical Study of the Nesting Level of Composite States within UML Statechart Diagrams. In Proceedings of 24th International Conference on Conceptual Modeling (ER 2005) Workshops, LNCS 3770, 12-22. 2005.
12. Denger, C., Ciolkowski, M.: High Quality Statecharts through Tailored, Perspective-Based Inspections. EUROMICRO 2003. pp. 316-325.
13. Evermann, J., Wand, Y. Toward Formalizing Domain Modeling Semantics in Language Syntax. IEEE Trans. on Soft. Eng., Vol 31(1). 2005.
14. ISO/IEC 9126. Software Product Evaluation-Quality Characteristics and Guidelines for their Use. Geneva.
15. Kitchenham, B., Pfleeger, S. L., Fenton, N.: Towards a Framework for Software Measurement Validation. IEEE Trans. on Softw. Eng., 21(12):929-944, 1995.
16. Object Management Group. MDA The OMG Model Driven Architecture. Available: http://www.omg.org./mda/, 2002.
17. Object Management Group. UML 2.0 OCL 2nd revised submission. OMG Document. Available at http://www.omg.org
18. Object Management Group. UML Specification Version 1.4.2, OMG Document formal04-07-02. Available at http://www.omg.org
19. Reynoso, L., Genero, M., Piattini, M. OCL2: Using OCL in the Formal Definition of OCL Expression Measures, In Proceedings of the 1st Workshop on Quality in Modeling QIM co-located with the ACM/IEEE MODELs 2006. 2006.
20. Saeki, M., Kaiya, H.. Model Metrics and Metrics of Model Transformation. In Proceedings of 1st Workshop on Quality in Modeling, pp. 31-45, Genova, Italy, Oct. 1, 2006.
21. Sunyé, G., Pollet, D., Traon, Y., Jézéquel, J.: Refactoring UML Models, UML'01: In Proceedings of the 4th Int. Conference on The Unified Modeling Language, Modeling Languages, Concepts, and Tools. 2001. pp.134--148.
22. Tang, M., Chen, M.: Measuring Object-Oriented Design Metrics from UML. In Proceedings of the UML 2002, LNCS 2460, pp. 368-382, 2002.

23. Vinter, R., Loomes, M., Kornbrot R.: Applying Software Metrics to Formal Specifications: A Cognitive Approach. In Proceedings of 5th. Int. Symposium on Softw. Metrics. March 20 - 21, 1998, pp. 216-223.
24. Briand, L. C., Morasca, S., Basili, V.R.: Property-Based Software Engineering Measurement, IEEE Trans. Softw. Eng. 22(1) 1996. pp.68--86, IEEE Press.
25. Reynoso, L., Rolón, E., Genero, M. And Garcia F., Piattini M.: Formal Definition of Measures for BPMN Models, IWSM/Mensura, pp. 285-306. Software Process and Product Measurement, Proceedings of the International. Conferences IWSM 2009 and Mensura 2009, Amsterdam, The Netherlands, 2009, Lecture Notes in Computer Science 5891.
26. Monperrus, M., Jézéquel, J., Champeau, J., Hoeltzener, B.: A Model-Driven Measurement Approach. MoDELS '08: Proceedings of the 11th international conference on Model Driven Engineering Languages and Systems. Pp 505-519. Springer-Verlag. 2008.
27. Genero, M., Piattini, M., Chaudron, M.: Quality of UML models. Information & Software Technology. 51(12), pp.1629-1630,2009.
28. Monperrus, M., Jézéquel, J.M., Champeau, J., Hoeltzener, B.: Measuring Models. Chapter in Model-Driven Software Development: Integrating Quality Assurance (Jörg Rech, Christian Bunse, eds.), IDEA Group, 2008.
29. Reynoso, L., Genero, M., Piattini, M.: Refinement and Extension of SMDM, a Method for Defining Valid Measures. J. UCS 16(21): 3210-3244 (2010).
30. Reynoso, L, Cruz-Lemus, J. A., Genero, M., Piattini, M.: Formal Definition of Measures for UML Statechart Diagrams using OCL. Proceedings of the 2008 ACM Symposium on Applied Computing, SAC 2008: 846-847
31. Reynoso, L., Amaolo, M., Dolz, D., Vaucheret, C., Álvarez, M. A Briand et al.'s framework-based UML/OCL Model for Increasing the Rigorousness of Measures Definition. 2013. Tech. Report UNC-UNPSJB. Available at: https://www.dropbox.com/sh/9vs6uy0re1owm8t/1tc2XIM3Yy/TechnicalReportFbPB.pdf

APPENDIX A: Formal definition of properties of Cohesion, Coupling, Complexity, Length.

This appendix lists the properties of *cohesion* and *coupling* of PbFB according to the Briand et al. framework [24]. Due to space limitations the *length* and *complexity* properties are not shown in this appendix but they are available in a technical report [31].

Properties of Cohesion and Coupling

These properties are meaningful when a modular system is defined:

```
context System::is_modular_system()
inv: self.getElements()->forall(e|self.modules
  ->exists(m |m.getElements()->includes(e) )
and self.modular->forall(m1, m2 |m1.are_disjoint(m2))
context System::cohesion_md()
pre: is_modular_system()

context System::coupling_md()
```

```
pre: is_modular_system()
```

A.1 Properties of Cohesion

Non-negativity and Normalization: The cohesion of a module of a modular system (or modular system) belongs to a specified interval **.**

```
context System::cohesion_md()
pre: is_modular_system()
    post non-negativity_normalizacion: result >= 0 and
result <= self.maxcohesion
```

Null Value: The cohesion of a module of modular system (or modular system) is null if is empty.

```
context System::cohesion_md()
inv null_value: self.getRelationships()->isEmpty()
          implies self.cohesion_md()=0
```

Monotonicity: Adding intra-module relationships does not decrease [module|modular system] cohesion. If there is no intra-module relationship among the elements of a (all) module(s), then the module (system) cohesion is null.

```
context System:: cohesion_md ()
inv monotonicity : self. cohesion_md() <=
self.module->any(true).addRelationship().cohesion_md()
```

Cohesive Modules: The cohesion of a [module|modular system] obtained by putting together two unrelated modules is not greater than the [maximum cohesion of the two original modules |the cohesion of the original modular system].

```
context System:add_modules(m:System)
post: result.relationships = re-
sult.getRelationships()@pre-
>including(m.getRelationships())
result.elements = result.getElements()@pre-
>including(m.getElements())

context System:: cohesion_md ()
inv cohesive_modules :
self.modules->exists(m1, m2 | are_notconnected(m1, m2:
System)
m1.add_modules(m2).cohesion_md() <=
m1.cohesion_md().max(m2.cohesion_md() )
```

A.2 Properties of Coupling

Non-negativity: The coupling of a [module|modular system] is non-negativity.

```
context System::coupling_md()
         inv non-negativity: result >= 0
```

Null Value: The coupling of a [module|modular system] is null if [OuterR(m)|R-IR] is empty.
We need two auxiliary functions, InputR(m) and OutputR(m). InputR(m) are the incoming relationships of a modular system whereas Output(m) are the outcoming relationships.

```
context System:: inputR():Set(Relation)
post: result = self.getRelationships()->collect( r |
self.getElements()->includes(r.target ) and
self.getElements()->excludes(r.source) )
context System:: outputR():Set(Relation)
post: result = self.getRelationships()->collect( r |
self.getElements()->includes(r.source ) and
self.getElements()->excludes(r.target) )

context System:: coupling_md()
body null_value: self.inputR()->isEmpty() implies cou-
pling_md() = 0 and
self.outputR()->isEmpty() implies coupling_md() = 0
```

Monotonicity: Adding inter-module relationships does not decrease coupling.

```
context System:: coupling_md()
inv monotonocity:
self.modules()->includes(m1, m2 |
self.are_not_connected(m1, m2) implies
self.coupling_md() <=
self.addRelationshipbetween_nonCC(m1, m2).coupling_md())
```

Merging Modules: The coupling of a [module|modular system] obtained by merging two modules is not greater than the [sum of the couplings of the two original modules|coupling of the original modular system], since the two modules may have common inter-module relationships.

```
context System:: coupling_md()
inv merging_modules :
    self.modules->forall(m1, m2 |
    m1.add_modules(m2).coupling_md() <=
    m1.coupling_md() + m2.coupling_md()  )
            self.getelements() = m1.getElements()
            ->including(m2.getElements())->flatten()
  implies self.length_md =
  m1.length_md().max( m2.length_md()))
```