Hindawi Publishing Corporation
Advances in Software Engineering
Volume 2013, Article ID 952178, 13 pages
http://dx.doi.org/10.1155/2013/952178

Hindawi

*Research Article*

# A Granular Hierarchical Multiview Metrics Suite for Statecharts Quality

**Mokhtar Beldjehem**

*University of Ottawa, School of Information Technology and Engineering, 800 King Edward, Ottawa, ON, Canada K1N 6N5*

Correspondence should be addressed to Mokhtar Beldjehem; mbeldjeh@uottawa.ca

Received 30 March 2013; Revised 7 June 2013; Accepted 7 June 2013

Academic Editor: Phillip A. Laplante

Copyright © 2013 Mokhtar Beldjehem. This is an open access article distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

This paper presents a bottom-up approach for a multiview measurement of statechart size, topological properties, and internal structural complexity for *understandability* prediction and assurance purposes. It tackles the problem at different conceptual depths or equivalently at several abstraction levels. The main idea is to study and evaluate a statechart at different levels of granulation corresponding to different conceptual depth levels or levels of details. The higher level corresponds to a flat process view diagram (depth = 0), the adequate upper depth limit is determined by the modelers according to the inherent complexity of the problem under study and the level of detail required for the situation at hand (it corresponds to the all states view). For purposes of measurement, we proceed using bottom-up strategy starting with all state view diagram, identifying and measuring its deepest composite states constituent parts and then gradually collapsing them to obtain the next intermediate view (we decrement depth) while aggregating measures incrementally, until reaching the flat process view diagram. To this goal we first identify, define, and derive a relevant metrics suite useful to predict the level of understandability and other quality aspects of a statechart, and then we propose a fuzzy rule-based system prototype for understandability prediction, assurance, and for validation purposes.

## 1. Introduction

The popular view of quality is still that of a subjective concept which perpetuates the idea that the more elaborate and complex product somehow offers a higher level of quality than its humbler counterpart. Whilst this misconception is well understood amongst "quality" professionals, the temptation remains to equate sophistication, instead of simplicity of function, with quality.

The term *software crisis* was coined in the late 1960 (NATO conference held in Garmisch-Partenkirchen, Germany, in 1968) [1] to refer to problems associated with software projects. These included budget and schedule overruns and problems with the *quality* and *reliability* of the delivered software. *Software quality* has been described as a complex and multifaceted concept, which basically means that it means different things to different people. Current approaches to assuring and measuring software quality are predominantly *process based* rather than *product based*. There are two major wrong assumptions here: (1) if we look after the process the product will look after itself, (2) quality must be built in the product (can only be assessed when complete and difficult to alter). These approaches allow the delivery of poor quality software in spite of seemingly wonderful quality management systems. As a result, the product may function far from smoothly, rating badly on a variety of quality attributes. Certainly those approaches were influenced by the manufacturing-based viewpoint of quality, and probably the main reason for this emphasis on procedure was based on the wrong assumption that the software product is notoriously difficult to assess for quality whereas the process appears much more amenable. So far, quality still an elusive feature of a software product and poor software quality continues to be an issue and a recurring theme in software development. Software quality aspects are captured as nonfunctional requirements and constraints, and thus on the contrary to functional requirement they are difficult to test. A design that fulfills all functional requirements but fails to meet nonfunctional requirement is not a valid design.

The essence of engineering quality software is to investigate the relationships among in-process metrics, project characteristics, and intermediate work products quality, and,

based on the findings, to engineer improvement in both process and product. Obviously, one needs procedures aimed at continually improving both process and product quality. A key point is that product as well as process metrics are essential if we wish to detect problems early in the development lifecycle, before they get out hand. Metrics thus can serve as early warning systems for potential problems. In order to achieve true indication of quality the intermediate work products must be subjected to measurement; that is, the internal attributes of quality must be related to specific external product quality requirements and quantified.

*Software measurement* during product development conjointly coupled with quality evaluation, prediction, and analysis of intermediate work products (or artifacts) at early stages of software development could detect modeling and design flaws, pinpoint potential trouble-spots, give advance warning of potential risks for focused remedial actions, provide clues that can lead to specific actions for improvement, provide semiautomated help to make informed design decisions in process, ease software design, guide the designer toward the most effective design, and could ensure a *preventive maintenance* layer (maintenance performed for the purpose of preventing problems before they occur), thereby improving the quality of software while reducing the cost of producing and maintaining software. This will materialize the idea of *design for quality*. Metrics need to be used throughout the development life cycle to gauge work products quality. How to approach such kind of measurement? How does one select the right metrics to gauge work products quality? Do metrics always suffice? Can metrics be selected from a bucket, or designed, allowing you to determine whether or not your work products satisfies your customers' quality requirement as they are expressed by their quality attribute set? What pitfalls to avoid in the process?

Modelers, as fallible humans with limited cognition, rarely, if ever, develop a perfect model to a complex problem on the first attempt. At best, they can hope for a crude beginning that through iteration and guided measurement, they can gradually refine and improve. To practice effective iteration, we must conceive models that are easy to create, easy to measure, easy to understand, and easy to revise. Iteration-based measurement is a practical reality.

*Measurement-driven predictive models* are good candidates to establish predictive causal relationships between external quality aspects and other internal indicator metrics based on historical data, provide early predictions of quality, and identify problems early so that timely actions can be taken to improve product quality, enabling software managers to prioritize and to better allocate resources. Product metrics are often used to predict various quality attributes of software products, such as *maintainability*, *testability*, *changeability*, and *error-proneness*. Various statistical analysis techniques and learning algorithms can be used to engineer and build such predictive models. The rational and main idea is that quality cannot be tested in a final product but must be built into its associated intermediate work products. The main goal is to help ensure that the *right* system is built and that it is *built right*.

This paper differs from other software quality approaches, because the paper's philosophy is that quality is at heart a technical problem (at least from the software engineer's perspective). Management plays a role by creating the right environment and adopting and conforming to standards, but the real attributes or aspects of quality, for example, simplicity, maintainability, testability, stability, usability, understandability, reusability, reliability, and modifiability and, the so-called "-ilities," are in the hands of the technical people. Design of software that is of high quality, easily extensible, and reusable is key requirements of any software project. A *model-driven programming* environment (such as Umple) has been touted as a promising emerging technique for achieving these software development goals, due to the possibility of code generation and reusability at the higher level of a model rather than at the level of executable code. As a result, modeling gains new importance and great visibility and scope of applicability in the context of model driven development (MDD). From modeling perspective understandability of the design is an important issue; some design choices may create very efficient systems but might be difficult to grasp (i.e., are ununderstandable) and thus should be avoided. A model should be accurate, easy to create, easy to understand, and easy to revise.

In this paper we examine aspects of statecharts quality and more specifically those connected to measuring, evaluating, predicting, analyzing, and assuring *understandability* of a statechart (the ease with which one might comprehend the inner workings as well as the behavior conveyed by a statechart at several levels of details) in order to restructure it, to fine-tune it, to refine it, to rework it, and/or to maintain it properly. As statecharts become increasingly large and complex, the need increases to predict and control their understandability. Statechart understandability is closely related to simplicity, transparency, compactness, conciseness, self-descriptiveness, structuredness, communicativeness, modularity and even resilience to change. Simplicity, keeps one's model understandable-A simple model is easier to understand, to test, to maintain and takes less development effort, this is known as the "*law of simplicity*." The main goal is to devise a statechart exhibiting the desired level of understandability. One has to design all software artifacts from inception to deployment with the understandability in mind. More specifically, we propose to investigate the issue of understandability from the angle of software measurement and from the modeler's viewpoint.

This paper is organized as follows. Section 2 surveys the related literature. Section 3 reviews briefly the statecharts visual formalism and introduces the idea of statecharts with measurable structure or topology. Section 4 describes and analyzes statecharts quality aspects. Section 5 proposes a granular framework for statecharts structural complexity and size metrics and then using the Goal-Question-Metric (GQM) paradigm defines a metrics suite for size, topological properties, and structural complexity of a statechart. Section 6 describes a fuzzy rule-based prediction system that might be used for validation purposes. Section 7 attempts to draw some conclusions and to suggest some research directions.

## 2. Related Work

The importance of a statechart model to be understandable to the software engineer (modeler, programmer, tester, and maintainer) is widely acknowledged in the software industry. To this goal, several approaches to assess and improve readability and understandability were attempted, and they broadly fall into three key categories: the proposal of new metrics for UML statechart diagrams, automatic layout of statecharts, and UML styles or diagramming guidelines.

In the first category, there have been a number of recent attempts at defining and validating metrics for UML Statechart. Metrics for behavioral models can be traced back to Derr [2], and a first proposal for transition complexity metrics such number of states, number of transitions, was described in connection with the OMT methodology Rumbaugh et al. [3].

Poels and Dedene [4] proposed structural metrics for event-driven object-oriented models. A metrics suite was defined based on the size, the structure, the dynamic behavior and the distance for object-oriented conceptual schemes.

Cartwright and Shepperd [5] proposed metrics and applied them to a software system of a telecommunication company and described an empirical investigation into an industrial object-oriented (OO) system comprising 133,000 lines of C++. The system was a subsystem of a telecommunications product and was developed using the Shlaer-Mellor method. They use 13 metrics (9 internal for the analysis phase and other 4 external that are all readily available early in the analysis and design stage) to build a prediction system for the size and defect density based on linear regression.

Carbone and Santucci [6] proposed metrics exploiting UML diagrams for effort estimation using the class complexity of an object oriented system. They defined metrics such as total number of states per class, total number of actions per class, and number of association per class For each class a class point (CP) value is computed, and CPs of all classes are aggregated in the use case, statechart, and interaction diagrams to compute the overall complexity of the system and estimate the project size.

Most notably is the work of Cruz-Lemus et al. [7], which also focused on metrics for statecharts diagrams and provided statistical validation of statecharts diagrams for understandability. They conclude the preliminary investigation mentioning that metrics such as number of activities, number of simple states, number of guards, and number of transitions seem to be highly correlated with the understanding of UML statechart diagrams.

In the second category as in Castelló et al. [8], layout and automatic layout of statecharts is another promising research area devoted to the visualization and presentation problems of statecharts in graphical editors. This is either to guide or to relieve the modeler the task and effort of organizing the statechart layout for readability. Even though those approaches deal with the presentation rather than the content, readability is closely related to understandability. An important issue those techniques must take into account is usability. Since statecharts need to be understandable and they must be compatible with the cognitive abilities of modelers (human being), the quest of tools that support visualization and navigation mode is needed, in particular the possibility to focus on a part of the statechart, to expand/collapse or zoom in/zoom out operations, or to the availability of an overall navigation map complemented by a detailed view. Animation too provides a rich environment for actively exploring statecharts. Multiple, dynamic, and graphical displays of a statechart at different level of details reveal properties that might otherwise be difficult to comprehend or even remain unnoticed. Moreover, animation makes it possible to picture and play with the statechart model.

In the third category, other efforts recommend following UML styles or diagramming guidelines as proposed by Ambler [9] (similar to programming guidelines) to increase the readability and thus the understandability of statechart diagrams. Within the UML modeling community, there are some modeling style conventions that have been widely agreed upon, because following them unequivocally leads to diagrams that are easier to read, understand, and maintain.

Our work stresses the importance of granular multi-view measurement of the statechart diagram at different depths rather than one measurement on a flattened diagram only. Moreover we propose a fuzzy rule-based system prototype for understandability prediction, assurance, and for validation purposes.

## 3. Statecharts Overview and Statecharts with Measurable Structure or Topology

Conceptual modeling is the cornerstone of software analysis and design. Models, the products of conceptual modeling, not only provide the abstractions required to facilitate mapping to code and communication between the analysts, programmers, testers, maintainers, project managers, stakeholders, and end users, but they also provide a formal basis for developing adequately new tools and techniques that will be used in the software technology. In MDD, models are not only supplementary artifacts for documentation or communication, but strict, formal abstractions of the software system to be developed Mellor [10]. Abstraction is a fundamental human capability that permits us to cope with complexity. It is worth stressing herein that modeling as well as meta-modeling is a core activity that is of utmost importance bearing in mind that conceptual modeling is by nature an abstraction and approximation of the reality. A model should contain only the essential and important details. The principle of "Occam's Razor" known as the *law of parsimony* [11, 12] states that "*Entities should not be multiplied unnecessarily*." In the context of statecharts entities might be states (both simple and composite), events, transitions, actions, and paths This principle is used extensively in modeling and in model simplification. The objective is to choose only those entities in the model which are absolutely necessary to explain the behavior of the world. Modeling as well as meta-modeling still is more an art than a science and remains a human activity which is error-prone. If models are abstractions built to understand a problem before implementing a solution, first of all they need to be understandable. Moreover, an ununderstandable model

is likely to difficult to revise, be difficult to test, and error-prone and can easily become too complex for humans to cope with.

As illustrated in Figure 1, there are several aspects of complexity and we broadly distinguish three of them, computational complexity (algorithmic complexity and concerns efficiency of computations), representational complexity (which reflects the syntax and semantic of the language used, which define the soundness of a language design—the absence of ill-defined statecharts in the UML in our study), and psychological complexity which is composed of three kinds of complexity, inherent complexity (or problem functional or behavioral complexity, which is problem-dependent and is thus beyond of the control of a modeler), cognitive complexity (the modeler's cognitive characteristics, limitations of the human cognition), and structural complexity (product/statechart complexity such as size metrics and McCabe's cyclomatic complexity metric). We limited the scope of our study and are interested in our investigation to statechart structural complexity and in particular at studying statechart structural metrics.

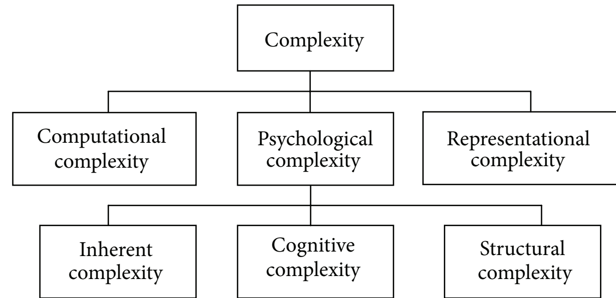



Figure 1: A Taxonomy of complexity metrics for software.

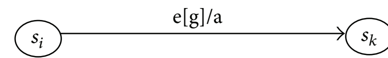



Figure 2: Basic transition in a statechart.

In general, it is difficult for humans to understand a complex model well. There is one important aspect in modeling; it is the restriction imposed on us by Miller's Law or the magical number *seven plus minus 2* principle, which stipulates that due to short-memory capacity limitations at any one time, we human beings are capable of concentrating on only approximately seven *chunks* (unit of information or *granules*) [13]. In addition, information held in the short-term memory decays after 18–30 seconds if not rehearsed [14]. However, a typical artifact, model, or diagram has far more than seven chunks. One way we humans handle this restriction on the amount of information we can handle at one time is to use *stepwise refinement*. This provides clues at devising readable and understandable models. How does Miller's work affect statecharts understanding? Every human (modeler) differs with respect to his or her short-term memory capabilities and every human differs in how he or she perceives complexity. Statechart that is complex from Miller's point of view can be several orders of magnitude more difficult to understand. This situation could exaggerate the individual differences among modelers with respect to short-term memory. Moreover, human communication and comprehension in the short-term are primarily sequential in nature. Obviously, Miller's Law in connection with modeling provides clues to a good design heuristic.

Statechart diagrams, in general, are visual formalism for describing dynamic aspects of complex systems behavior; they were proposed by Harel [15, 16] to model the control requirements of complex *reactive systems* (control-driven, or event-driven systems in contrary to *transformational* systems). Statecharts extend the modeling power of basic finite state model in several ways to overcome the basic machine's scalability and concurrency limitations. Scalability is usually accomplished by partitioning or hierarchic abstraction. As described succinctly in [15], "statecharts = state diagrams + depth + orthogonality + broadcast communication." That is, statecharts extend basic state model semantics, adding hierarchic nesting of states (depth, which reflects the *insideness*), the ability to model two or more independent control strategies (orthogonality, to cope with concurrency), and allowing any transition output action to generate an internal event for any other transition (broadcast) that might cause *chain reaction* of a any length, that is, an event seen everywhere in the diagram at the same time. Basically, statecharts add more expressive power and serve to reduce complexity of state diagrams by structuring them and avoiding the combinatorial explosion that has plagued them. They offer a contour-based notation for state diagrams as a special case of general diagram notation called *higraphs* (high graph, hierarchical graph, or topological graph). State diagrams as well as statechart diagrams exhibiting high structural complexity can be rendered less complex through partitioning and hierarchic decomposition to equivalent statecharts diagrams. The initial state is shown by a filled circle and the transitions are shown in the form of e[g]/a. As illustrated in Figure 2, the typical meaning of the form e[g]/a is that "the transition is taken when the triggering event e occurs and the guard condition(s) g holds, this transition results into firing the action(s) a."

In the context of object-oriented paradigm, statecharts describe the behaviour of an object. All classes defined in the class diagram have an associated statechart; each object behaves according to the statechart associated to its class. Statecharts represent a natural choice for object behavioral modeling. This is essentially due to built-in features that enforce modularity and control complexity. In practice, these extensions make statecharts suitable in situations where simple state diagrams would become cluttered. Complex systems typically contain much redundancy that structuring mechanisms can simplify.

A *state* is a particular configuration of the values of data attributes. What is observable about a state is a difference in behavior from one state to another; that is, if the same message is sent to the object twice it may behave differently depending on the state of the object when the message is received. A *transition*, or change, from one state to another is triggered by an event, which is typically a message arrival. A guard is a condition that must hold before the transition can be made. *Guards* are useful for representing transitions

to various states based on a single event. An action specifies processing to be done while the transition is in progress. Each state may specify an *activity* to perform while a transition is in progress.

A state associated with multiples (two or more) outgoing transitions is called a *decision* state (or a coordination center) because a decision is made at this node to select a transition to follow in actual execution according to the event that occurs when this node is active. It is also called a *branching* state for obvious reasons. Similarly, a state associated with multiple incoming transitions is called a *junction* state. In a statechart, typically a state might be simultaneously a disjunction and junction state. The number of outgoing transitions from a state reflects the degree of local navigability from it to reach direct target states.

The notion of a statechart has been adopted and extended by the Open Management Group (OMG) (http://www.omg.org/) as one of the specification formalisms of the Unified Modeling Language (UML) (http://www .uml.org/), and also by W3C in the form of SCXML (http://www .w3.org/TR/scxml/) which, was originally targeted at specifying voice. Statecharts are usually presented in a graphical form, although there are standard "textual" formats, including SCXML, and XML Metadata Interchange (XMI) (http://www.oasis-open.org/cover/xmi.html) for UML.

## 4. Statecharts Quality Aspects

It has become common knowledge that the early identification and resolution of potential problems can significantly reduce development time and cost and at the same time increase the quality of the overall software product [17, 18]. It is widely accepted that structural diagrams (such as class diagram) have a great influence on the quality of software. Several proposals of metrics suites have been proposed in the literature for such diagrams (class diagrams). However, there have been little effort directed toward the behavioral diagrams (use case, statecharts, sequence, and collaboration diagrams), and as a result they have been relatively overlooked in the software measurement. MDA (OMG Model-Driven Architecture) (http://www.omg.org/mda/) has been a prominent means to enhance the understandability of the system's structure and behavior. MDD shifts the focus of software development from programming to modeling. It has prompted industries to develop tools which can generate the code from the model in high level languages like C, C++, PHP, or JAVA. Therefore ensuring model's understandability becomes highly essential for the software industry.

We point out herein that the definition of new metrics that capture dynamics aspects of OO is an interesting topic of investigation. Our focus will be on those related to statecharts as they constitute inputs to code generators in MDA environments. In MDA (OMG Model Driven Architecture) (http://www.omg.org/mda/), assuming that transformation rules are implemented correctly, predicting as well as evaluating the quality of the statechart model becomes a central issue.

Software quality also depends on one's point of view; the criteria relevant for end users in judging software quality differ from those applied by software engineers who have to design, develop, test, and maintain software. Based on these one has to distinguish between external and internal factors of software quality. According to Boehm [19] *understandability* is the extent to which the software is easily comprehended with regard to purpose and structure, and he pointed out that the factors influencing maintainability are testability, *understandability*, and modifiability.

ISO-9126 (ISO, 2001) [20] provides a hierarchical framework for quality definition, organized into quality characteristics (or factors) and subcharacteristics. As illustrated in Figure 3, there are six top-level quality characteristics, with each associated with its own exclusive (nonoverlapping) sub-characteristics: (1) functionality (suitability, accuracy, interoperability, and security), (2) reliability (maturity, fault tolerance, and recoverability), (3) usability (*understandability*, learnability, operability), (4) efficiency (time behavior and resource behavior), (5) maintainability (analyzability, changeability, stability, and testability), (6) portability (adaptability, installability, conformance, replaceability). Certain tradeoffs have to be made since some factors are synergistic in nature, and others are potentially conflictive. Thus, increased efficiency should not be achieved at the price of a lower understandability.

*Usability* is defined as a set of attributes that bear on the effort needed for use, and on the individual assessment of such use, by a stated or implied set of users. *Understandability* is defined as the capability of the software product to enable the user to understand whether the software is suitable, and how it can be used for particular tasks and conditions of use.

In the context of statecharts modeling users are modelers, coders, testers and maintainers, and even code generators (*codifiers* or programs) of particular interest to us herein is the subcharacteristic understandability of usability characteristic and more specifically understandability of statecharts diagrams. Intuitively, it is difficult to restructure, rework, and maintain a statechart diagram whose behavior you do not understand. A statechart diagram which is simple enough to understand its behavior is a desirable one. Understandable statecharts models are more malleable than codes, and model changeability is not considered an issue and might be taken for granted. Symptoms of bad statecharts design might include, to name just few, opacity (i.e., hard to understand), unstructuredness (duplication of transitions, intense presence of undesirable subtle, pathological transitions, and highly cluttered and bad structure), complexity, size (i.e., large size), rigidity (i.e., hard to change), fragility (i.e., easy to break), and lack of reusability (hard to reuse), and s.o. As a good practice, it is worth giving up some performance on noncritical operations if you can devise and use an understandable statechart. For this reason we need to fine-tune and restructure a statechart to ensure the level of desired understandability. Unless one has a performance problem, it is not worth doing a lot of optimizing, because one risks making a model harder to understand. Moreover, a highly optimized statechart often sacrifices ease of change and makes a model difficult to maintain the difficulty of understanding a model limits its reusability too. It is therefore important to devise statecharts that exhibit the desired level

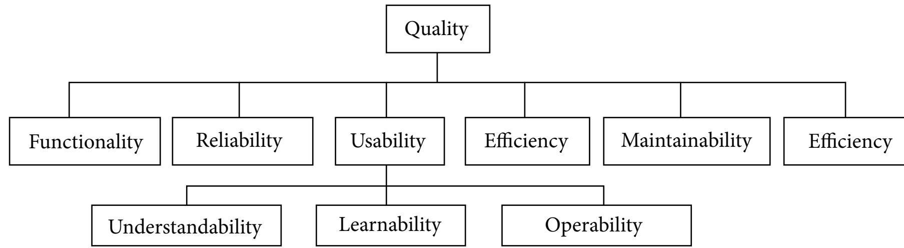



Figure 3: The software quality hierarchy.

of understandability. Despite this, there are few guidelines on how to construct understandable statecharts models, and we notice a lack of tool support in the field too.

The Goal-Question-Metric (GQM) paradigm of Basili and Rombach [21] (which is in fact just a partial methodology) will be adopted and used as approach to metrics modeling. It is rigorous goal oriented approach to measurement in which goals, questions, and measurement are closely related. The *goals* (conceptual level) are first identified, and then *questions* (the operational level) that relate to the achievement of the goal are identified, and for each question a *metric* (quantitative level) that gives objective answer to the particular question is identified. The statement of the goal is very precise, and the goal is related to individuals or groups. Consider our study whose goal is the prediction of the quality and more precisely understandability of statecharts diagrams through internal quality indicators or measures. There are several valid questions which may be asked at this stage, and these are questions which require answers to determine the extent to which the understandability is achieved: who are the involved professional? individuals (such students, professors, modelers, programmers, testers, and maintainers) that use the statechart and what is their level of experience, One might define a metric level of experience such as average year of experience. Each question is then analyzed to determine the best approach to obtain an objective answer to the question and to identify which metrics are needed and the data that needs to be gathered to answer the question objectively. Metrics are the objective measurements to give a quantitative answer to the particular questions. The questions and measurements are thereby closely related to the achievement of the goal (understandability of a statechart) and provide an objective picture of the extent to which the goal is being satisfied.

The objective of measurement is to improve the understanding and prediction of statechart understandability. GQM leads to focused measurements which are related to the goal (quality and understandability of a statechart) rather than measurement for the sake of measurement.

Predicting the level of understandability of the statechart control structure is thus of paramount importance. While the high-level statechart diagram is partitioned and/or decomposed hierarchically into low-level substatecharts or statecharts subdiagrams, attention must be given to the complexity of the organization of the subdiagrams with regard to their relationships with each other (or *topological properties*) and their internal attributes for quality. Attention must be given to the evaluation of their structural complexity too. A statechart is understandable if someone else other than the creator can understand the diagram (as well as the creator after a time lapse). Minor improvements to efficiency are not worth compromising understandability. The challenge is to devise statecharts at "right" abstraction level according to the desirable understandability level. From the modeler's perspective, understandability is the ability of the modeler to build a logically accurate mental model of the statechart model that she or he is utilizing (reading, revising, fine-tuning, testing, and maintaining) to achieve his or her dynamic goal.

The internal structure of a statechart diagram is a matter of great interest to us at the design stage. Informally, a statechart diagram that contains a lot of actions but not too many states with higher outdegrees will be very easy to understand and design. It will present little opportunity for the designer to stray from the straight and narrow. Another statechart diagram with fewer actions but a really complex control structure will offer the designer plenty of opportunity to introduce faults. Informally, a state whose Outdegree (resp., indegree) is very large is probably good candidates for examination and rework. Obviously, as a statechart needs to be human readable as well as machine readable, a cluttered statechart diagram is less desirable than an uncluttered transparent self-contained diagram. As such it needs to be compatible with the modeler's cognitive limits too. The "understandability" of a model for a machine shows up in error-free compilation. For human modelers, model understandability must be explored, evaluated, and predicted. One cannot measure directly and quantify understandability as it is a multidimensional fuzzy concept, contrary to the definition of conventional crisp binary concept of understandability; it is allowed herein in our modeling to be granulated and graduated, that is, be a matter of degree, and as such we deal with it and represent it in terms of levels of understandability by the means of linguistic values in terms of labels (Zero, Low, Medium, and High) interpreted by either membership functions and/or possibility distributions as illustrated in Figure 4. Due to the effect of granulation, the transition from understandability to ununderstandability is gradual rather than abrupt. "Understandability is Zero," means it is not understandable. Doing so will ease during the prediction phase the identification of those statecharts with the values Zero and Low and prioritize to rework them first in order to improve and achieve the desired level of their understandability. Depending on time and budget

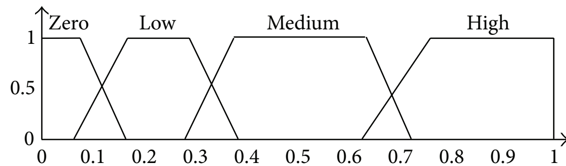



Figure 4: Understandability represented as a fuzzy linguistic variable.

constraints, we might be interested too at enhancing those with the Medium value.

In practice the modeler's *chunking* and *tracing* capabilities are used to analyze and comprehend a statechart, and this is not always possible due to the cognitive complexity and more specifically to the modeler's cognitive inherent limits (small short-term memory capacity) to deal with complex *hypertext-like topological* structures of statechart. In general, due to the topological modular nature of the statechart, it is not necessary for the modeler to understand an entire statechart in order to understand just one part of it (a composite state, or a chunk). Informally, to ensure understandability, one needs to devise a statechart diagram whose complexity matches or is just slightly below the (human) modelers' cognitive capacity by devising a manageable number of self-contained meaningful chunks that have reasonable sizes and depths, easy to follow (or traverse) transitions at an acceptable levels of structural complexity. Informally, an understandable statechart should allow a modeler to analyze and comprehend it at any level of abstraction without having to understand the detail or lower-level states.

As a matter of perception, large statecharts are generally (but not necessarily) more complex than small statecharts. Informally, an understandable statechart should be composed of understandable constituent parts (or chunks), and conversely only understandable constituent parts (chunks) might compose an understandable statechart. A *chunk* is a group of related elements (states, transitions, events, and actions) and is formed around the state under focus. A chunck is the *object under measurement* (OUM). This leads us to adopt a multi-view measurement to assess a statechart at different view levels.

The complexity involved in the management and description of large statecharts can be faced by partitioning the overall collection of the composing entities into smaller, more manageable, units. Chunks (cohesive units that are loosely connected with each other) as a conceptual construct offer a general grouping mechanism that can be used to decompose a given statechart diagram into subdiagrams and to provide a meaningful separate behavioral description for each of them.

The focus herein is on inherent characteristics of the statechart itself in the hope that controlling these internal quality indicators will result in improved external product quality such as understandability; namely, this is known as measurement-driven *design for understandability*. Better structure and reduced complexity make statecharts diagrams easier to read and understand and thus to extend and reuse. As a good practice, it is recommended that we devise statecharts with ease of understanding in mind.

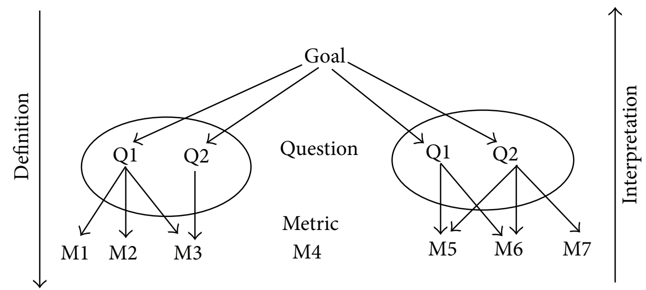



Figure 5: The GQM paradigm of Basili and Weiss.

Statecharts quality measurement-driven prediction models in general and fuzzy models in particular could constitute an interesting solution to software quality by providing some preventive maintenance layer on software at its early stages of its life cycle during modeling. The earlier understandability can be predicted, the better it is from the standpoint of statechart design and software development.

## 5. A Granular Framework for Statecharts Structural Complexity and Size Metrics

Understandability metrics are needed to verify if the desired understandability level is achieved prior to the passage to next phases (code generation). In departure from the conventional approaches that deal with only a flat statechart and firstly start by applying a flattening procedure, we attempt herein to deal with a statechart diagram at different conceptual depths or equivalently at several abstraction levels. More specifically it consists to evaluate at different granularity levels constructs (simple state, composite state, all states level diagram) corresponding to different conceptual depths exposing different views, and subsequently to aggregate their measurements. The main idea is to adopt a multi-view measurement and to study and evaluate a statechart at different levels of granulation corresponding to different conceptual depth levels or level of details. The higher level process view corresponds to a flat statechart (depth = 0), and the adequate upper depth limit is determined by the modelers according to the inherent complexity of the problem under study and the level of detail required for the situation at hand (it corresponds to the all states view). The rationale is to keep during modeling a manageable complexity. For purposes of measurement, we proceed using *bottom-up* strategy starting with all state view diagram (see Figure 6 for more details), identifying and measuring its deepest composite states constituent parts (or chunks) and then gradually collapsing them to obtain the next intermediate views (we decrement depth), until reaching the high-level flat process view diagram.

As shown in Figure 5, GQM main idea is that measurement ought to be carried out for purpose, and that is only within the context of purpose or goal that a metric may be determined to be useful.

Table 1: Metrics suite for a statechart (at the state and at the statechart diagram levels).

| Metric name | Metric definition |
|---|---|
| Indegree I(s) of a state s (fan-in) | The total number of entering transitions into s |
| Outdegree O(s) of a state s (fan-out) | The total number of exiting transitions from s |
| NSS(s) number of direct source states into s | The total number of states that are direct source states of entering transitions into s |
| NTS(s) number of direct target states from s | The total number of states that are direct target states of exiting transitions from s |
| Number of actions associated to the state NA(s) | The total number of distinct actions associated to the state that exists on outgoing transitions from this state under focus |
| Number of events associated to the state NE(s) | The total number of distinct events associated to the state that exists on outgoing transitions from this state under focus |
| Number of states NS(s) | The total number of states including simple states as well as composite states that are the constituent parts of the substatechart under measurement |
| Number of states at a single depth level NSL(sc) | The total number of states including simple states as well as composite states that are the constituent parts of all the substatecharts viewed at a single given depth level |
| Conceptual depth of the statechart CD(sc) | The conceptual depth of the statechart CD(sc), which is the *maximum* value of conceptual depths of all states that are the constituent parts of the statechart. Due to nesting levels and overlapping, we take the maximum |
| Number of internal transitions NIT(s) | The total number of transitions where both the source state and the target state are within the enclosure of the composite state s. Outgoing and incoming transitions to the composite state s under study are ignored |
| Number of Interlevel intrahierarchy external transitions NIET(s) | The total number of external interlevel transitions crossing two or more levels in the state hierarchy, from or into s |
| Number of deeper external cross-hierarchy transitions NDET(s) | Number of deeper cross-hierarchy transitions relative to depth(s) crossing in other parts of the statechart, from or into s |
| (NIET(s) + NDET(s))/((I(s) + O(s)) | A ratio ranging in [0, 1] |
| McCabe's cyclomatic complexity CC(s) | CC(s) = NIT(s) − NS(s) + 2 |

Using the GQM [21–23] template for goal definition, the goal pursued with the definition of the metrics for statecharts diagrams is as follows.

Analyze: Statechart diagrams.

With the aim of: Predicting and controlling.

In relation to: Quality and more specifically understandability.

From the point of view of: The modelers.

Based on the input of modelers, this measurement application goal was represented as the following list of questions.

(1) What factors influence the level of understandability of a statechart diagram?

(2) What is the contribution of size (number of states, of transitions, of events, and of actions) and the conceptual depth of a statechart to its understandability?

(3) What is the contribution of the other topological properties (*insideness*, *connectedness*, and *adjacency*) and the internal structure of a statechart to its understandability?

(4) What is the contribution of the structural complexity of the statechart to its understandability?

(5) What are possible indicators and standards on which prediction of satisfactory levels of understandability can be based?

The metrics suite we propose is shown in Table 1. Both primitive (counts of directly measurable characteristics such as number of transitions) and derived metrics (computed by mathematical combinations of two or more primitive metrics such as McCabe's cyclomatic complexity) are included. These metrics were defined to measure statecharts size, topological relationships (enclosure, connectedness, and adjacency), and structural complexity based on intrinsic properties and building blocks of statecharts (state diagrams, depth, orthogonality, and broadcast communication) and graph theory and more specifically higraphs (high graphs, hierarchical graphs, or topological graphs) on which a statechart is build on. These metrics are objective (i.e., there is an agreed procedure for assigning values to the metrics, computed by counting), they are empirical (i.e., the data is obtained by observation), they are easy to collect and could be computed in the early phases, providing thresholds that could be used for judgments, and they are programming language independent. Conventionally, a software quality metric is defined as a function which inputs software data and outputs a single value interpretable as the degree to which software possesses an attribute that

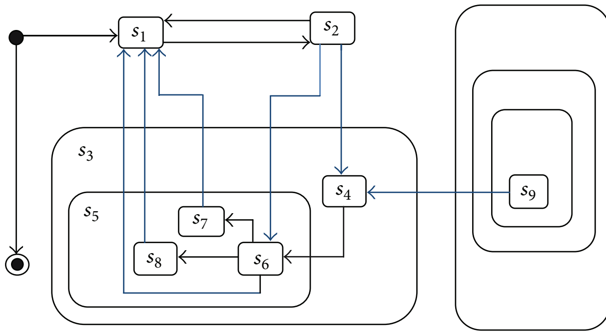



Figure 6: All states view diagram highlighting intra hierarchy interlevel and cross-hierarchy transitions.

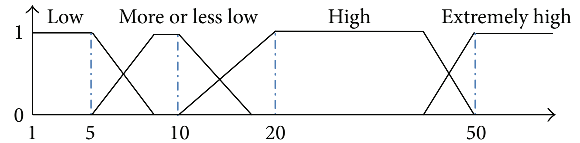



Figure 7: McCabe's cyclomatic complexity represented as a linguistic fuzzy variable.

affects quality. In departure of conventional methods, we assume herein that a software quality metric is a linguistic variable that might have linguistic values represented by labels of Zadeh [24, 25] fuzzy sets (such exteremely small, very small, more or less small, medium, more or less large, very large, exteremely large) and interpreted by membership functions as illustrated in Figure 7. Thus each metric is interpreted by a fuzzy partition or equivalently a fuzzy sequence. Graduation and granulation are used conjointly in modeling a fuzzy metric on a *fuzzy ordinal scale*.

As illustrated in Figure 6, the composite state $s_3$ is made up the composite state $s_5$ and the simple state $s_4$, the composite state $s_5$ itself is made up the simple states $s_6$, $s_7$, and $s_8$. Typical interlevel transitions on the same hierarchy are those between $s_8$ and $s_1$, $s_7$ and $s_1$, $s_6$ and $s_1$, a typical interhierarchy transition is between $s_9$ and $s_4$. The conceptual depth of the statechart equals three.

*5.1. Top-Level Process View (Coarse-Grained Measurement).* At the process view, as in a finite state machine one has an unstructured higher level diagram. At this view, all states might be thought of conceptually as simple states, that is, there are no composite states to deal with or to account for at this level. Proposed metrics are illustrated in Table 1. McCabe's cyclomatic complexity [26] for the corresponding planar graph is computed straightforwardly (using the formula $e - n + 2$, where $e$ is the number of transitions, and $n$ the number of states). McCabe's cyclomatic complexity appears to represent structural complexities of statecharts in a manner that parallels the ways Millers says humans perceive complexity; that is, the cyclomatic complexity of a statechart can be taken as an indication of the diagram's understandability when studied by humans (modelers). McCabe's metric is really an indication of the number of linearly independent (control) paths through a diagram, and reflects procedural complexity. A statechart with cyclomatic complexity greater than eight is too complex for human (modeler) mind to understand without placing some of the details into long-memory. From a path perspective a complexity greater than 10 indicates that the diagram is too complicated because there are too many paths to be stored and simultaneously processed in the modeler's short-term memory. Consequently, understanding the diagram will be unnecessarily difficult. Designing statecharts that have low structural complexity will certainly do a lot toward easing their understandability. In practice the upper limit has been set as fifteen. A compromise of ten has been selected because it is just beyond the short-term memory's upper limit, and is the base of our decimal numbering system. Difficulty of understanding a statechart is largely determined by complexity of control structure of diagram. Understandability declines with increasing cyclomatic complexity. In connection with modeling, cyclomatic complexity provides clues to a good design heuristic.

As illustrated in Figure 7, presumably statecharts with cyclomatic complexity value of Low are generally considered simple and easy to understand (i.e., understandability is High), of More or less Low value (moderately complex) are considered not too difficult to understand (i.e., understandability is Medium), if High, the complexity is perceived as high (very complex) and thus are considered difficult to understand (or understandability is Low). When the value is Extremely High, the statechart is overly complex and for practical purposes becomes ununderstandable (not understandable, i.e., understandability is Zero). Of course, we assume that cyclomatic complexity alone is not sufficient to determine the level of understandability of a statechart. Note that the depth of state $s$ might remain unchanged or might keep varying (increasing) while design proceeds depending on the involvement of the state $s$, as long as partitioning and hierarchical abstraction mechanisms are used or concurrency is introduced by means of orthogonality, this is irrespective that we are using *top-down*, *bottom-up* design strategy or a *mixed* strategy in which the two strategies are combined. Using a top-down design strategy, one way to organize a model is to start by having a high-level diagram (process view) with subdiagrams expanding certain states gradually using *stepwise refinement*, each of the subdiagrams further expanded and this process continues until the resulting elements are in easy comprehensible form and straightforward to work with, thus obtaining a well-structured statechart diagram at a certain adequate conceptual depth level (sketching all states view). Using a bottom up, one may start with partitioning (or clustering) states using commonalities to obtain high abstract level aggregate composite states. Accordingly, a mixed strategy is therefore highly recommended in practice to devise a statechart properly. For cognitive considerations, the conceptual depth must be kept manageable, and thus we need to set up an upper limit of the maximum depth of hierarchy (too many nested levels of compound states limit understandability).

As shown in Figure 8, presumably statecharts with depth value of Small are generally considered simple and easy to

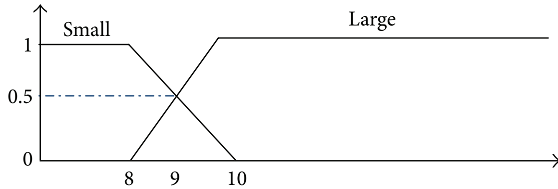



Figure 8: The state hierarchy nesting level (conceptual depth).

understand. When the depth value is Large, the statechart for practical purposes becomes ununderstandable. Of course, we assume that depth alone is not sufficient to determine the level of understandability of a statechart.

*5.2. Chunk Level (Intermediate-Grained Measurement).* There may exist one or more intermediate views between the flat view and the all states view of the statechart diagram. A composite (aggregate or superstate) state might be thought of as an abstract state, and thus the metrics for a simple state might be used to deal with it at an adequate level of abstraction. Composite states might be obtained either by the means of the partitioning in terms of aggregated states and hierarchical abstraction mechanisms in terms of nested states or by the means of orthogonality in terms of concurrent composite states. In addition, composite states do have their specific metrics as illustrated in Table 1. The computation of all proposed metrics is straightforward except for the McCabe's cyclomatic complexity, and thus we have adapted and extended it to deal with single-entry/multiple-exit, multiple-entry/single-exit, and multientry/multiexit statecharts and concurrent composite states. A good well-structured self-contained composite state should convey a well-identified behavior and should be examined, assessed and understood by itself to a large extent (without regards for what operations do, what they operate on, or how they are implemented). Basically, hierarchical abstraction has to do with pulling out common characteristics (such as states having same transition's event/action to same target state) of a group of states into a higher-level composite or superstate with a factorized transition, thereby defining behavioral blocks of higher-level while reducing duplication and redundancy. There are a variety of reasons to for keeping composite states "*black boxes*" (hiding the internal workings or behavior), but the primary one is to minimize the ripple caused by maintenance changes. When connecting source and target states to a composite state, the modeler should not care about how the composite state behaves-only that the desired result occurs. Valid and meaningful abstractions facilitate understanding.

The complexity involved in the management and description of large statecharts can be reduced by partitioning the overall collection of the composing entities (states and transitions) into smaller, more manageable, control units. Chunks (cohesive units that are loosely connected with each other). A chunck as a conceptual construct offer a general grouping mechanism that can be used to decompose a given statechart diagram into sub-diagrams and to provide a meaningful separate behavioral description for each of them. Individual chunks (or subdiagrams) represent fragments of behavior, while the entire statechart diagram represents integrated behavior. See Figure 9 for more details. The level of abstraction is reduced gradually in a process known as *stepwise refinement*, with more and more detail introduced at each step, the behavioral description at each step becoming a more low-level specification of the model. This process continues until the design is framed in a satisfying structure. Good statecharts designs that are easy to build and easy to understand and change are based on a bunch of chunks, each of which is "cohesive," or well-glued together, and only loosely "coupled" to other chunks. Chunk cohesion is a measure of chunk "wholeness" and coupling measure interdependence among chunks.

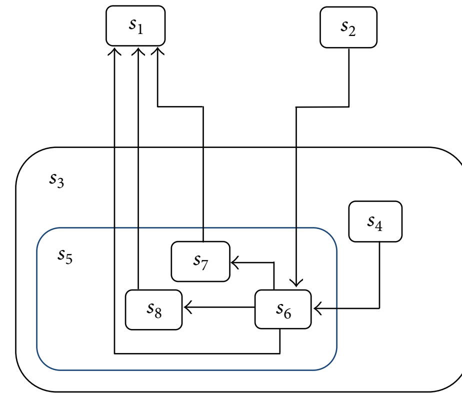



Figure 9: A conceptual *chunk* formed around the composite *state under focus* $s_5$.

This way of abstracting statecharts for measurement purposes is in the spirit of modularization and *information hiding* in the sense of Parnas [27] and is too closely related to *chunking* in *schemata theory* as described in cognitive psychology and advocated by Gray [28].

The crux of the prediction problem is then, to select a *minimal set* of "*critical*" metrics to predict accurately the statechart understandability level. This is a heuristic and to some extent a trial-and-error process, where one has to try collecting some metrics, determine if they are useful, replace the ones that are not useful. Depending on the accuracy required of desirable level of understandability, one may define the number of the entries in the set and establish criteria for the initial entries in the set.

Concurrency can be modeled with a *product machine*. The states of a product machine are combinations of states of two or more basic machines. Product state behavior is implicit in a statechart. In a product machine, every possible combination of individual states is an explicit product state. Concurrency comes in three flavors: (1) aggregation concurrency, (2) concurrency within object, and (3) synchronization of concurrent activities. In either case, in practice a statechart with two or more concurrent regions is first transformed to one or more equivalent statecharts in the product machine. In case we have $p$ independent connected components, $SC_1, SC_2, \ldots, SC_p$, one might use the formula $e - n + 2p$,

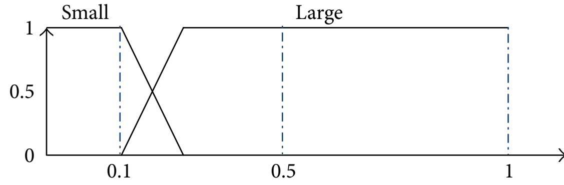



Figure 10: The computed ratio metric (NIET(s) + NDET(s))/((I(s) + O(s)).

where $e$ is the number of transitions, $n$ the number of states, and $p$ the number of components in the total graph, or compute summation of their complexities as McCabe's cyclomatic complexity is additive with respect to the union by using the following formula:

$$\mathrm{CC}\left(\mathrm{SC}_1 \cup \mathrm{SC}_2 \cdots \cup \mathrm{SC}_p\right) = \mathrm{CC}\left(\mathrm{SC}_1\right) + \mathrm{CC}\left(\mathrm{SC}_2\right) + \cdots \mathrm{CC}\left(\mathrm{SC}_p\right). \quad (1)$$

*5.3. All States View (Fine-Grained Measurement).* As illustrated in Figure 6, all simple states as well as all composite states including their enclosures are sketched in this all states view. By collapsing composite states one might obtain or recover the corresponding previous intermediate views gradually. We use this bottom up strategy for measurement purposes; that is, we start analyzing and measuring the characteristics of those deeper constructs (chunks) forming the statechart at this view, and then we collapse them to obtain previous intermediate view to measure (corresponding to a lower depth, we decrease the depth), then iterate gradually, while aggregating measures incrementally until reaching the higher level process view statechart. The computed ration metric (NIET(s) + NDET(s))/(I(s) + O(s)) should be kept as small as possible as it limits the *modular topology* of the statechart. Ideally, it is better that the modeler does not recourse to using those inter-level and cross-hierarchy transitions. However, it seems that in practice some modeling situations dictate their usage.

As shown in Figure 10, presumably statecharts with ratio metric value of Small are generally considered simple and easy to understand. When the value is Large, the statechart for practical purposes tends to become ununderstandable. Of course, we assume that the ratio metric alone is not sufficient to determine the level of understandability of a statechart. If we assume that we have a common e[g]/a for the three transitions on the same hierarchy between $s_8$ and $s_1$, $s_7$ and $s_1$, $s_5$ and $s_1$, $s_6$ becomes a default start state in $s_5$, and $s_4$ becomes a default start in $s_3$. As illustrated in Figure 11 we obtain an equivalent statechart, thereby improving the understandability while preserving the behavior of the previous one. The point is to make this refactoring behavior-preserving transformation and other ones automatic.

## 6. A Fuzzy Rule-Based Prediction System

In general, software engineering activities are knowledge intensive and software modeling and design is a good

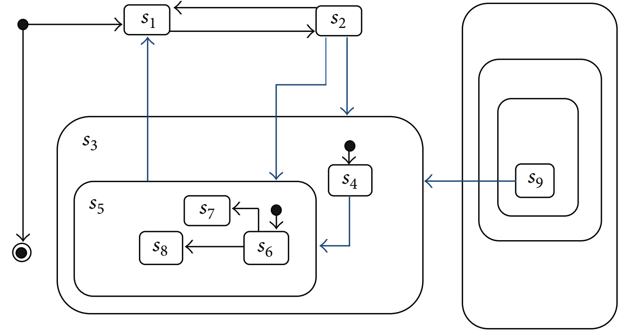



Figure 11: An equivalent all states view obtained after refactoring of the statechart in Figure 6.

Table 2: The rule-base of the fuzzy rule-based prediction system.

| CC | Depth | Ratio | Understandability |
|---|---|---|---|
| Extremely high | Large | Large | Zero |
| Extremely high | Large | Small | Zero |
| Extremely high | Small | Large | Zero |
| Extremely high | Small | Small | Low |
| High | Large | Large | Zero |
| High | Large | Small | Low |
| High | Small | Large | Low |
| High | Small | Small | Medium |
| More or less low | Large | Large | Low |
| More or less low | Large | Small | Medium |
| More or less low | Small | Large | Medium |
| More or less low | Small | Small | High |
| Low | Large | Large | Low |
| Low | Large | Small | High |
| Low | Small | Large | Medium |
| Low | Small | Small | High |

application area since the knowledge is generally heuristic in nature and software engineers tend to think on terms of rules and more specifically on terms of fuzzy rules. Independent variables (internal metric indicators) are measured to predict the dependent variable (external level of understandability). In order to capture those causal relationship between internal metrics and the levels of understandability, we attempted to build a rule-based fuzzy predictive prototype for the "*proof of concept*" based on some statecharts design best practices, *design heuristics* related to our defined metrics. We obtained the following rule base (a collection of fuzzy *if-then* rules shown in Table 2). Each row represents an *if-then* fuzzy production rule, for instance, row seven corresponds to the following rule seven ($R_7$) as follows:

> $R_7$: *if* (CC is High) AND (Depth is Small) AND (Ratio is Large) *then* Understandability is Low.

Knowing the values of metrics of a given statechart, an *inference engine* will be able to compute the understandability level by first firing adequate rules of the rule base and then applying an appropriate standard *defuzzification* procedure (such as the center of gravity method). This system may be

thought of as a tool that disseminates best practices. The most powerful contribution by such a system is to put at the service of the inexperienced modeler, the best practices, design heuristics, "*rules-of-thumb*," experience, and accumulated wisdom of the best modelers. This is no small contribution as it empowers designers to capture their thinking in a coherent and comprehensive fashion in the framework of *human-oriented* software engineering, thus enabling them to produce understandable statecharts structures that are faster, smaller, simpler, cleaner, and produced with less effort. Those structures will be the building blocks of high quality software systems that exhibit a robust and resilient architecture.

## 7. Conclusion and Future Work

Producing elegant statechart that is efficient while still being easily understood by their peers is still a true art form. The originality and contributions of our framework proposed in this paper includes as follows:

(i) a human-oriented framework for statechart understandability based on the interplay between soft computing, cognitive psychology, and software engineering;

(ii) a granular hierarchical multi-view metrics suite for statecharts quality instead of a crisp binary one-level view of a flattened statechart diagram;

(iii) a novel bottom-up approach for a multi-view measurement of statechart size, topological properties and internal structural complexity for statechart understandability prediction and assurance purposes;

(iv) new design heuristics based on new metrics and valid cognitive principles;

(v) indicators and standards on which prediction of satisfactory levels of statechart understandability can be based?;

(vi) a fuzzy rule-based statechart understandability prediction system;

(vii) a soft computing human-oriented framework for studying, analyzing and assuring statechart understandability.

Software quality is of increasing importance as the use of software become pervasive. Statecharts, like class diagrams are the key conceptual artifacts of the early development, so focusing in their quality should contribute to the overall quality of the software product which is ultimately implemented. Most interesting systems are not most conveniently modeled as statecharts. We have argued that a satechart with measurable structure or topology allows the prediction of the level of understandability early at design time. These measures have to be used judiciously, but they are easy to obtain and give internal indicators of understandability. Ultimately detecting low level of understandability (or lack of it) requires to rework (or *to refactor* either manually or automatically) the control structure of the statechart in order to cope with and reduce the complexity, while preserving the behavior and improving the understandability. By "complexity," I mean "needless or unnecessary complexity." The main goal is to achieve automated support for modeling understandable statecharts. This leads to improved quality and understandability of systems whose complexity exceeds the intellectual (cognitive) capacity of a single modeler or team of modelers.

We also need to define the analysis model of all indicators. Then, we plan to perform a principal components analysis [29] on the proposed metrics to select the *minimal set* of "critical" independent metrics that influence the understandability of a statechart to allow suitable higher prediction accuracy. With all the metrics already tested and validated, we need to search for relevant indicators for *understandability*, *learnability,* and *operability*, and, ultimately, a unified set of indicators for statechart *usability*. Finally, all metrics and indicators need to be progressively refined as we plan for more complete case studies and controlled experiments. The aim is to find not only usability related problems in statecharts, but also appropriate solutions for these problems and produce a list of behavior-preserving transformations which, if executed either manually, semiautomatically, or automatically would lead to better understandability, usability, and quality of the statechart. With software integrated into so many critical parts of modern day-to-day life, modelers must be very aware of quality.

Future work includes better tool support for this approach, as well as mechanisms to collect automatically metrics data from statecharts, to analyze data statistically, and to learn and simulate fuzzy systems and statecharts, a high integration of all steps of the entire process into a single adequate tool. To this goal, learning automatically a rule-based fuzzy system from a *repository* of statecharts models with measurable structures examples by using machine learning algorithms and/or soft computing is too a good option to explore [30, 31]. It is hoped that the paper will stimulate further work in a field whose importance will increasingly be recognized.

## References

[1] P. Naur and B. Randell, Eds., *Software Engineering: Report on A Conference Sponsored by the NATO Science Committee1968*, Scientific Affairs Division, Brussels, Belgium, 1969.

[2] K. Derr, *Applying OMT*, SIGS Books, Prentice Hall, New York, NY, USA, 1995.

[3] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen, *Object-Oriented Modeling and Design*, Prentice Hall, Englewood Cliffs, NJ, USA, 1991.

[4] G. Poels and G. Dedene, "Measures for assesing dynamic complexity of aspects of object-oriented conceptual schemes," in *Proceeding of the 19th International Conference on Conceptual Modeling (ER '00)*, pp. 499–512, 2000.

[5] M. Cartwright and M. Shepperd, "An empirical investigation of an object-oriented software system," *IEEE Transactions on Software Engineering*, vol. 26, no. 8, pp. 786–796, 2000.

[6] M. Carbone and G. Santucci, "Fast & Serious : a UML based metric for effort estimation," in *Proceeding of the 6th International ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering (QAOOSE '02)*, pp. 35–44, 2000.

[7] J. A. Cruz-Lemus, M. Genero, S. Morasca, and M. Piattini, “Using practitioners for assessing the understandability of UML statechart diagrams with composite states,” *Lecture Notes in Computer Science*, vol. 4802, pp. 213–222, 2007.

[8] R. Castelló, R. Mili, and I. G. Tollis, “A framework for the static and interactive visualization of statecharts,” *Journal of Graph Algorithms and Applications*, vol. 6, no. 3, pp. 313–351, 2002.

[9] S. W. Ambler, *The Element of UML 2.0 Style*, Cambridge University Press, Cambridge, UK, 2005.

[10] S. J. Mellor, *MDA Distilled: Principles of Model-Driven Architecture*, Addison-Wesley, Boston, Mass, USA, 2002.

[11] W. M. Thorburn, “Occam’s razor,” *Mind*, vol. 22, no. 4, pp. 287–288, 1913.

[12] W. M. Thorburn, “The Myth of Occam’s razor,” *Mind*, vol. 27, no. 3, pp. 345–353, 1918.

[13] G. A. Miller, “The magical number seven, plus or minus two: some limits on our capacity for processing information,” *Psychological Review*, vol. 63, no. 2, pp. 81–97, 1956.

[14] W. J. Tracz, “Computer programming and the human thought process,” *Software*, vol. 9, no. 2, pp. 127–137, 1979.

[15] D. Harel, “Statecharts: a visual formalism for complex systems,” *Science of Computer Programming*, vol. 8, no. 3, pp. 231–274, 1987.

[16] D. Harel, A. Pnueli, J. Schmidt, and R. Sherman, “On the formal semantics of statecharts,” in *Proceedings of the 2nd IEEE Symposium on Logic in Computer Science*, pp. 54–64, Ithaca, New York, NY, USA, 1987.

[17] B. W. Boehm, *Software Engineering Economics*, Prentice Hall, New York, NY, USA, 1981.

[18] B. W. Boehm, “Improving Software Productivity,” *IEEE Computer*, vol. 20, no. 9, pp. 43–57, 1987.

[19] B. W. Boehm, J. R. Brown, H. Kaspar, M. Lipow, G. J. MacLeod, and M. J. Merrit, *Characteristics of Software Quality*, TRW series of software technology, Elsevier Science, North-Holland, The Netherlands, 1978.

[20] ISO/IEC and 9126-1:2001, “Software engineering—product quality—part 1: quality model,” 2001.

[21] V. R. Basili and H. D. Rombach, “The TAME project :towards improvement-oriented software environment,” *IEEE Transactions on Software Engineering*, vol. 14, no. 6, pp. 758–773, 1988.

[22] V. Basili and D. A. Weiss, “Methodology for Collecting Valid Software Engineering Data,” *IEEE Transactions on Software Engineering*, vol. 25, no. 4, pp. 456–473, 1984.

[23] R. van Solingen and E. Berghout, *The Goal/Question/Metric Method: A Practical Guide For Quality Improvement of Software Development*, McGraw-Hill, New York, NY, USA, 1999.

[24] L. A. Zadeh, “Fuzzy sets,” *Information and Control*, vol. 8, no. 3, pp. 338–353, 1965.

[25] L. Zadeh, “The concept of linguistic variable and its applications to approximate reasoning part I,” *Information Sciences*, vol. 8, pp. 199–249, 1973.

[26] T. J. McCabe, “A complexity measure,” *IEEE Transactions on Software Engineering*, vol. 2, no. 4, pp. 308–320, 1976.

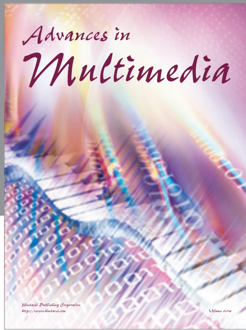
[27] D. L. Parnas, “On the criteria to be used in decomposing systems into modules,” *Communications of the ACM*, vol. 15, no. 12, pp. 1053–1058, 1972.

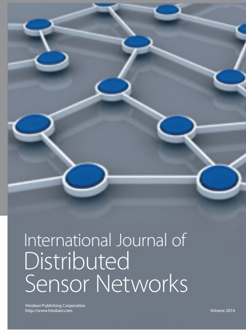
[28] P. Gray, *Psychology*, Worth Publishers, London, UK, 2007.

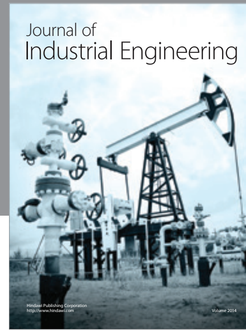
[29] D. Dunteman, *Principal Component Analysis*, Sage University Press, Thousand Oaks, CA, USA, 1989.

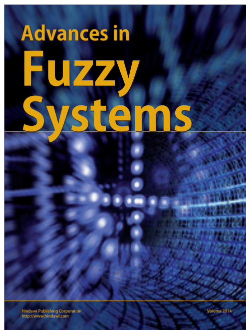
[30] M. Beldjehem, “A granular unified min-max fuzzy-neuro framework for learning fuzzy systems,” *Journal of Advanced Computational Intelligence and Intelligent Informatics*, vol. 13, no. 5, pp. 520–528, 2009.

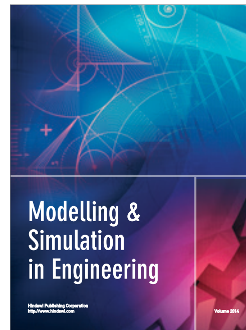
[31] M. Beldjehem, “A granular unified hybrid MinMax fuzzy-neuro framework for predicting and understanding software quality,” *International Journal of Software Engineering and Applications*, vol. 4, no. 4, pp. 17–36, 2010.












Hindawi

Submit your manuscripts at
http://www.hindawi.com