Metric Framework for Object-Oriented Real-Time Systems Specification Languages*

P. Nesi*   M. Campanai!

*Department of Systems and Informatics, Faculty of Engineering, University of Florence
Via S. Marta 3, 50139 Firenze, Italy, tel.: +39-55-4796265, email: nesi@ingfi1.ing.unifi.it

!CESVIT, Center for Software Quality, CQ_ware, Fortezza da Basso, Palazzina Lorenese, Viale Strozzi, Firenze

September 26, 1994

Abstract

A framework for maintaining under control and analyzing object-oriented system specifications of real-time systems by using a set of metrics covering technical, cognitive and process-oriented views is presented. The indicators defined can be used for monitoring the evolution of system quality and for effort prediction. The use of metrics for the estimation of reusability, verifiability and testability is analyzed. The metric framework is integrated in a CASE tool named TOOMS, which is based on TROL, a dual object-oriented language with both descriptive and operational capabilities. TOOMS allows to describe the system at different levels of structural abstractions, and at different levels of specification detail such as many other languages and models for real-time systems (e.g., OSDL, ObjectTime, ObjectChart). According to this, the metrics proposed are capable of producing estimations at each level of system specification, thus allowing the incremental specification/metrication. The metric framework must be regarded as a support for controlling the process of software development in order to guarantee the final quality.

Index terms: analysis and design metrics, measurement, reactive systems, method integration, object-oriented model, reuse, IO-based, CASE tool.

1 Introduction

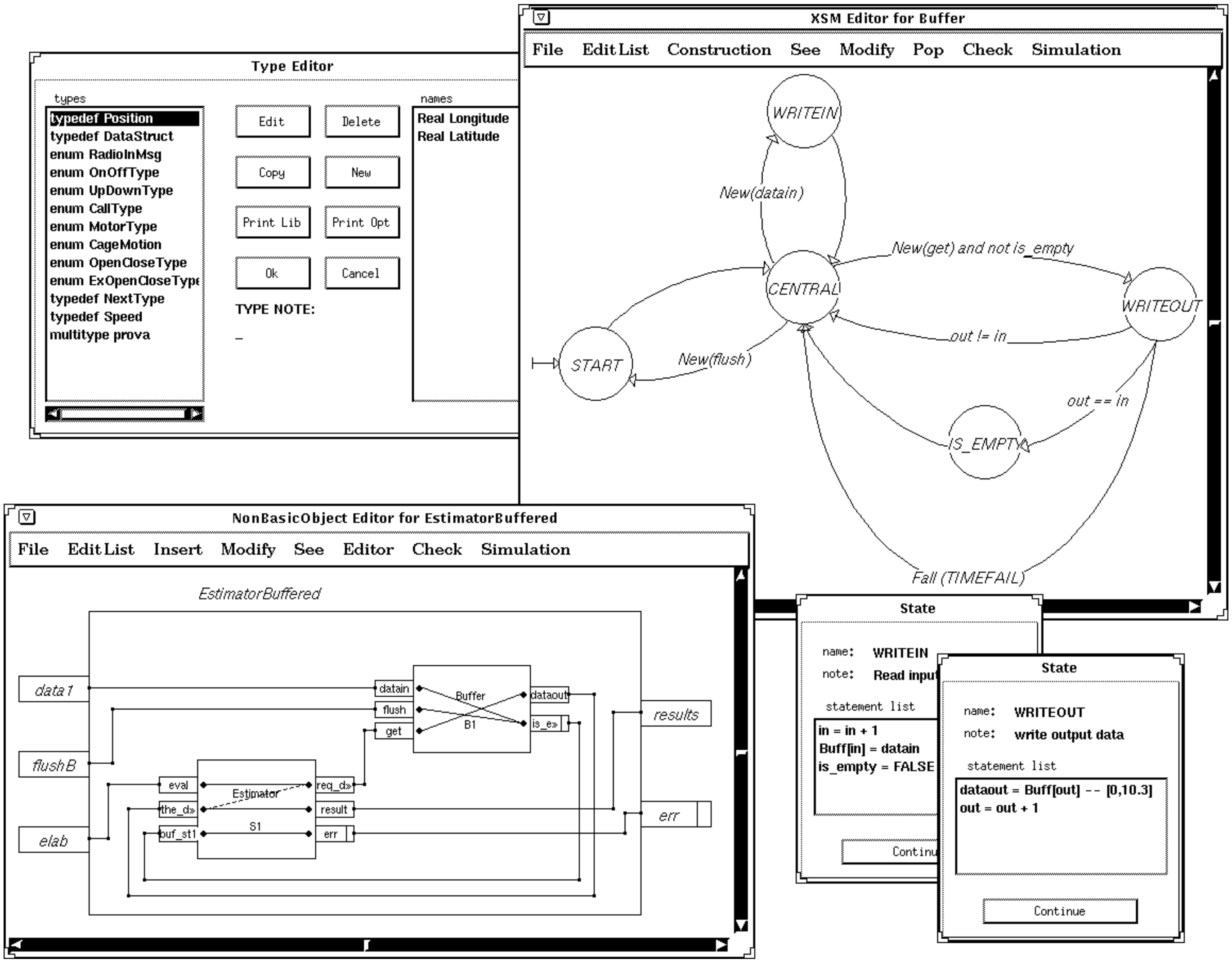
There is an over-abundance of metrics in the literature. Complexity and size are often defined on the basis of available metrics; many attempts to metrics validation have been made, but unfortunately for each positive validation there is a negative one. Some studies with metrics and measurement frameworks for object-oriented systems have been presented in [Laranjeira 1990], [Meyer 1990], [Panaroni et al. 1990], [Henderson-Sellers 1993], [Yap et al. 1993], [Coulunge et al. 1993], [Li et al. 1993], [Jensen 1991] where general concepts for the estimation of system size, complexity and reuse level have been proposed. Furthermore, an analysis of the impact of the *"reuse, as the most enticing promise"*, in the object-oriented system development has been carried out in [Henderson-Sellers 1993]. However, it should be noted that in the measurement of object-oriented systems, a general poor design and a lack of adherence to proper measurement principles are still present. This lack is also due to the different definitions of what the object-oriented paradigm is. Moreover, several formal definitions of the correct measurement principles

*This work was partially supported by the Italian Research Council, CNR (Consiglio Nazionale delle Ricerche) have been presented (e.g., [Fenton 1991], [ESPRIT-5494 1992], [Zuse 1994]), but they are still oriented to the structured approach. In addition, it should be noted that in order to define a specific framework for measuring the quality of the specifications given in languages suitable for real-time system specification, many specific metrics must be defined [Warburton 1983], [Jensen et al. 1985], [Wearing 1992]. This is due to several factors, (a) the need of using particular languages for specifying real-time systems (e.g., TROL [Bucci et al. 1994], TRIO+ [Morzenti et al. 1992], OSDL [Braek et al. 1993], etc., as can be observed in [Bucci et al. 1995]), (b) the need to give more evidence to system behavior – i.e., to the presence of temporal and logical constraints on system behavior, (c) the need of verifiability metrication of the specification consistency of the system under analysis, (d) the need of obtaining valid measurements even if the system is only partially specified (this is very useful for many algebraic and logical languages such as Object-Z [Carrington et al. 1990], VDM++ [Dürr et al. 1992], TRIO+ [Morzenti et al. 1992], TROL [Bucci et al. 1994], etc.).

Recently, a growing attention on the process of software development has created the need to get process-oriented information and to integrate metrics into the software development process. Furthermore, owing to the presence of many differences among projects by the same company, it is important to create an integrated environment to perform project-oriented *tailored* measures. This means that, it is important for a company to adopt a unique method and approach for project measurement, but is approach must be capable to be tuned in order to adapt its features to different types of projects and languages. This process of adaption is usually performed by using adjusting weights and thresholds [Henderson-Sellers et al. 1994].

In this paper, a shift on prospective is presented (a) in the definition of metrics, (b) in the integration of metrics in the development process, by providing an integrated framework for system measurements. In fact, the metrics proposed have been integrated in TOOMS, which is a CASE tool for the specification of real-time systems [Bucci et al. 1993]. The structure of the TOOMS tool is comprised of a set of visual editors and utilities. In TOOMS, the user interacts with the visual editors describing the system under specification by means of few graphical symbols. In particular, the Block Editor is adopted to specify *non-basic object classes*, which model the system decomposition/composition defining a class a set of communicating sub-objects; while the Machine Editor is used to specify *basic object classes* as extended state machines (i.e., XCMs and XSMs), for modeling classes which cannot be further decomposed (see Fig.1); and, the Type Editor to define new data types [Bucci et al. 1993] (see App.A). Utilities, for providing metrication, verification, validation, and simulation are directly integrated into the visual editors. A code generator, which transforms the system specification from the TROL language into C++ code, is also available. TOOMS is based on an object-oriented formal language and model named TROL [Nesi 1993], [Bucci et al. 1994]. TROL adopts a dual model which is capable of integrating the operational and the descriptive formalism; its operational or descriptive features are similar to other IO-based object-oriented specification languages and methods – e.g., TRIO+ [Morzenti et al. 1992], ObjectCharts [Coleman et al. 1992], Object-Oriented version of SDL (OSDL) [Braek et al. 1993], Object-Time [NorthernTelecom 1993]. Most of these approaches have the capabilities of specifying system behavior, structure and functionality, and allow the verification and validation of composition/decomposition mechanisms. In addition, the incremental specification of system description is allowed; where classes are defined by means of their external and internal class descriptions (which correspond to public and private class members according to the object-oriented paradigm).

Even if the metric framework proposed in this paper has been defined for the TOOMS/TROL model it can also be applied to the above-mentioned formalisms with minor changes, since most of the metrics



typedef Position
typedef DataStruct
enum RadioInMsg
enum OnOffType
enum UpDownType
enum CallType
enum MotorType
enum CageMotion
enum OpenCloseType
enum ExOpenCloseTyp
typedef NextType
typedef Speed
multitype prova

Real Longitude
Real Latitude

TYPE NOTE:

name: WRITEIN
note: Read input

statement list

```
in = in + 1
Buff[in] = datain
is_empty = FALSE
```

name: WRITEOUT
note: write output data

statement list

```
dataout = Buff[out] -- [0,10.3]
out = out + 1
```

Figure 1: Some Visual Editors of the TOOMS tool.

proposed are independent of the programming language as will be shown later. The metric model was integrated in the TOOMS tool in order to maintain under control the evolution (i.e., quality and effort) of real-time systems under specification; thus, quantitative parameters and related metrics have been defined for evaluating all aspects of the system under specification with a greater attention to the specification quality, reusability, verifiability, testability and cost prediction.

The paper is organized as follows. In Sect.2, the metrics which have been integrated in the TOOMS tool are discussed. The validation of the most important metrics proposed is presented in Sect.3. The mapping of the general concepts with respect to the metrics proposed together with the discussion about the language dependency, and methodologies support are reported in Sect.4. In the same section, the quality model is presented together with a discussion about the suggested guidelines. Conclusions are drawn in Sect.5. To improve paper readability, in App.A the main features of the TOOMS/TROL notation are reported, while detailed descriptions of the TOOMS tool and TROL languages can be found in [Bucci et al. 1993], and [Bucci et al. 1994], [Nesi 1993], respectively.

2 Metric Framework of TOOMS

In this section, the metrics of the metric framework integrated in the TOOMS tool are presented. The metrics proposed have been defined in order to give the TOOMS/TROL users a reasonable help for maintaining under control and evaluating their work. The aim of the integration is to cover the entire life-cycle of the system under specification by using a unique tool in order to provide software developers and managers appropriate instruments for establishing software quality since the early stages of system development. In specification languages, such as OSDL, TROL, TRIO+, etc., also partially defined specifications can be validated and tested (by proving properties through mathematics or by simulation). In Fig.2, different phases of the specification life-cycle are reported, where the system can be simulated and metricated. In TROL, as in other languages, when a class is only described by means of its external class description (services and clauses or assertions) the simulation can be performed by using clauses or other high-level description of class behavior. Analogously, the class complexity and size can be approximately predicted by using only the information available in the external class description (complexity and size). For these reasons, the model proposed allows the application of metrics at each level of specification detail, and thus the system development can be continuously maintained under metrication (see Fig.3). Therefore, metrics are used as a prevision model, as well as to control the software quality (see Sect.4.3).

In the literature, many metrics frameworks have been presented – e.g., [Henderson-Sellers 1991], [Zuse 1994]. Our approach is based on three different system views: a *technical*, a *cognitive*, and a *process-oriented* view. The *technical* view refers to the software engineering aspects of system specification (size, complexity, etc.); the *cognitive* view takes into account the external understandability and verifiability of system components and libraries; and, the *process-oriented* view refers to system aspects that are influenced by or can influence the process of system development (productivity, reuse, size, cost, etc.). These three views are evaluated in a common measurement framework in which each view can influence the others. Metrics of each view can be used in different phases of system evolution: the *cognitive* metrics during system development and/or in system maintenance, the *technical* metrics for the evaluation/certification of some specific characteristics of the system; the *process-oriented* metrics for evaluating the impact of technology on the whole development process. All these measurements should be estimated even if the system under development is not yet completely specified according to the above-mentioned specification

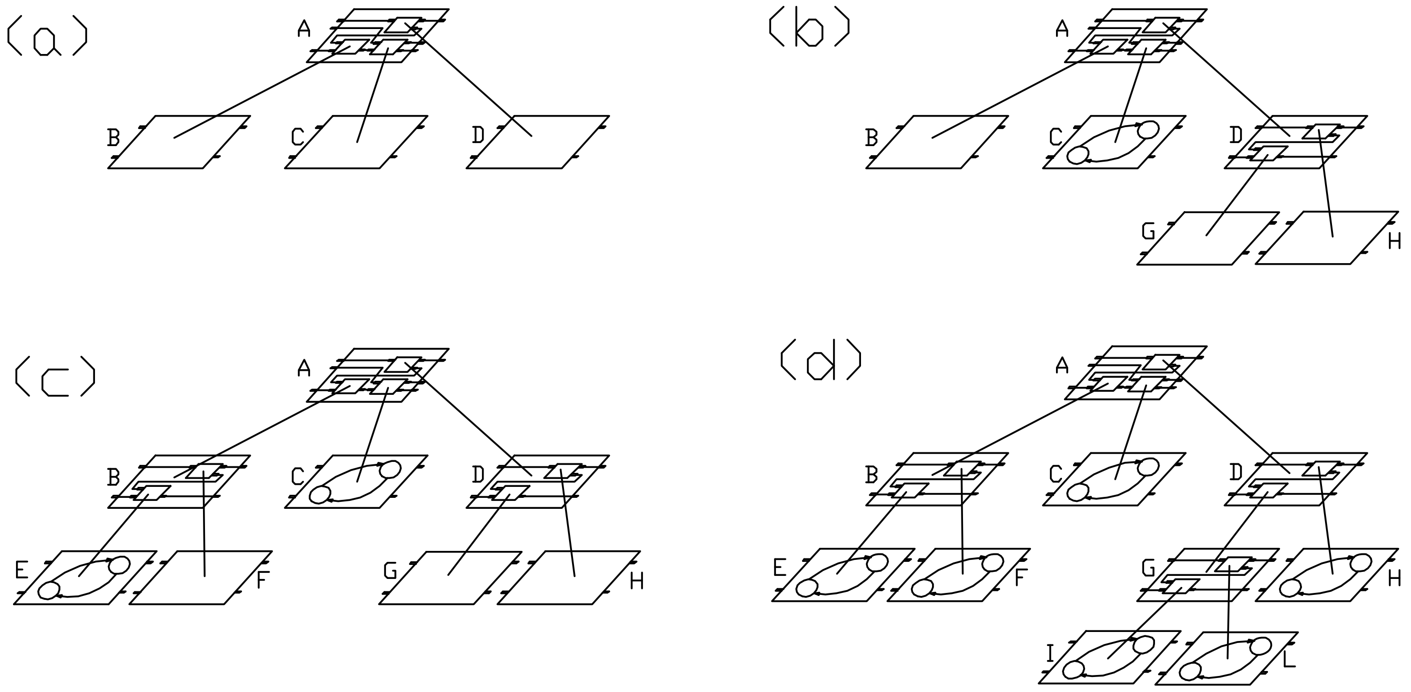


Figure 2: Different phases of system specification. In (a), A is defined and implemented as a non-basic object class, while for B, C, and D the only external class description has been defined. In (b), C has been implemented as an extended state machine (i.e., XSM) while D as a non-basic object class, etc.

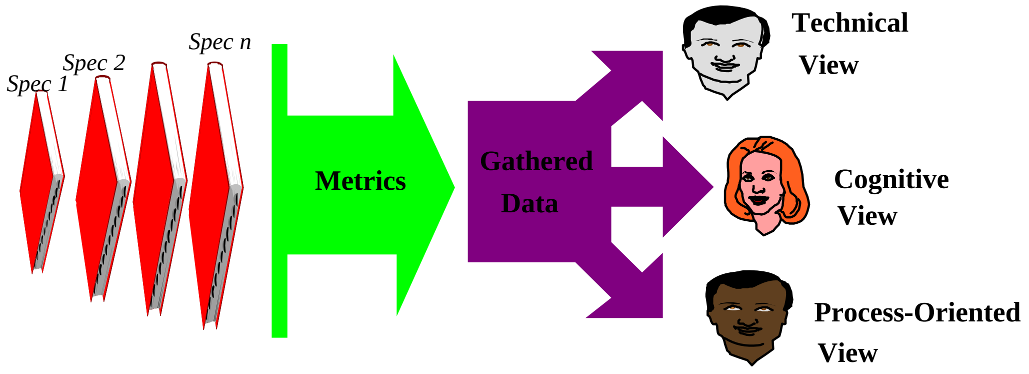


Figure 3: A continuous approach to quality improvement through metrics.

languages for real-time systems. In object-oriented systems development, a greater effort is required in the design phase, and considerably less for program maintenance. For these reasons, the metrication of system specifications is very important since the early stages of system design.

2.1 Technical Metrics

In this section, technical metrics to evaluate the class complexity and size are reported. For non-concurrent systems, the class complexity can be easily related to class size, while in the case of real-time systems (where the system behavior is defined by that of many concurrent processes, frequently modeled by means of state machines), these two concepts can be very different. Most of the traditional technical metrics for the estimation of size or complexity have been defined in order to predict the system cost [Fenton 1991], [Laranjeira 1990]. These measurements of size are mainly based on the estimation of the number of Lines Of Code (LOC). In the case of formal languages for the specification of real-time systems, the LOC estimation is not a suitable measure since the dynamic aspects of the system under measurement are neglected (the corrections which are usually performed on the basis of the development context and on the kind of application are hardly applicable to formal specification languages for real-time systems). In fact, it is likely that two specifications having the same size can be very different in cost if they are implemented by using a different number of communicating processes. In addition, traditional methods for measuring system size do not take into account the presence of temporal constraints or assertions [Thomas et al. 1989] – i.e., clauses for describing the class or system behavior. For this reason, more specific technical metrics have been proposed.

Given a class c, the following symbols will be used in the rest of the paper: NRS_c number of required services, NPS_c number of provided services, NA_c number of attributes, NC_c number of clauses, and NP_c number of paths (for the meaning of "path" see App.A.2). The above numbers also include the features inherited from the superclasses of the class under consideration according to the object-oriented paradigm. In the case of a method-based approach, the number of paths corresponds approximately to the number of concurrent class methods or class processes (as in OSDL). A list of symbols adopted in this paper is reported in Tab.5 of Sect.4 to improve the paper readability.

2.1.1 Class Complexity, CC

Following the object-oriented paradigm the estimation of class complexity of the system under analysis corresponds to the estimation of the class complexity of the class which models the system itself. The typical methods to estimate the complexity which are based on fan-in and fan-out such as those in [Card et al. 1990] are in this context (i.e., I/O-based object-oriented specification languages for real-time systems) less suitable, since the input/output complexity is only a small part of the whole class complexity. Therefore, the Class Complexity (CC) can be seen as the weighted sum of the Internal and the External Class Complexity (ICC and ECC):

$$CC = w_{ICC}\ ICC + w_{ECC}\ ECC$$

where w_{ICC}, and w_{ECC} are tailorable weights. Note that, since ICC and ECC take into account the complexity of the class by considering all its superclasses, then CC also includes such complexities.

The class complexity can be very useful to evaluate the system maintainability and reliability [Li et al. 1993]. In addition, basic object classes should have a complexity lower than a given threshold to ensure a selected software quality in terms of satisfying maintainability and reliability constraints.

2.1.2 Internal Class Complexity, ICC

The Internal Class Complexity (ICC) takes into account the Class Path Complexity (CPC) and the Class Attribute Complexity (CAC). In addition, the ICC also describes how the *Paths* of a given class are internally coupled in using class attributes by means of the term $CCPC$, (Class Coupling Path Complexity). The ICC is also influenced by the intrinsic concurrency of the class (Class Internal Task Complexity, $CITC$) – e.g., a class comprised of only one *Path* is obviously less complex than the same class implemented with two *Paths* (i.e., which correspond to two processes). The object-oriented design usually consists in a process of class decomposition and/or object composition for defining new classes. At this level, the Class Path Complexity has no meaning since the paths are not yet implemented. On the contrary, in the design phase, it is very important to evaluate the class complexity on the basis of the exchange information by the class attributes (which in turn are instances of some classes). This fact is measured with the Class Internal Communication Complexity, $CICC$.

Therefore, the internal class complexity has been defined as:

$$ICC = w_{CAC}\ CAC + w_{CPC}\ CPC + w_{CCPC}\ CCPC + w_{CITC}\ CITC + w_{CICC}\ CICC$$

where w_{CAC}, w_{CPC}, w_{CICC}, w_{CCPC} and w_{CITC} are application-dependent weights. In the following, details about the above terms are given.

Class Attribute Complexity, CAC

The complexity of each attribute (Attribute Complexity, AC) is equal to the class complexity of the attribute itself, thus $AC = CC$. Hence, the class attribute complexity, CAC, is obtained by considering all the class attributes:

$$CAC = \sum_{i}^{NA} AC_i$$

In many object-oriented languages, such as in C++, there is a set of predefined basic types which are not implemented as classes. For these basic types the class complexity CC must be predefined. In TROL, a suitable class has been defined for each basic type [Bucci et al. 1993].

Class Path Complexity, CPC

The Class Path Complexity is estimated by using the expression:

$$CPC = \sum_{j}^{NP} PTC_j$$

where PTC is the Path Complexity defined as:

$$PTC_j = w_{PMC}\ PMC_j + w_{PEC}\ PEC_j + w_{PIC}\ PIC_j$$

where: PMC_j is the sum of McCabe-like complexities of the statements which are present in the class path j [McCabe 1976], [Henderson-Sellers 1992]; PEC_j is the path external complexity which estimates the cost of the external procedure calls, considering the calls which are present in path j:

$$PEC_j = \sum_{i}^{AllCallsIn(j)} CallComplexity(i,j)$$

where the $CallComplexity(i,j)$ is evaluated by considering the complexity of the procedure parameters (the term 'call' means the use of a function from the library); PIC is the path internal complexity which estimates the cost of the access to the class attributes. The PIC for path j holds:

$$PIC_j = \sum_{i}^{NA} AC_i \; USE(i,j)$$

where $USE(i,j)$ assumes the following value: 0 if the attribute i is not used by path j, 1 if it is only read, and 2 if it is read and/or written; and AC is the already defined Attribute Complexity.

In TROL, the concept of Path is present only for basic object classes; therefore, for non-basic object classes $NP = 0$ and, thus, the CPC is equal to zero. The weights w_{PMC}, w_{PEC}, and w_{PIC} mainly depend on the application field under analysis. In the case of real-time system specification, they are assumed to be equal to 1, 1, and 2, respectively. In this way, a higher importance is given to the costs of attribute sharing among internal processes (i.e., paths) of an object.

Class Coupling Path Complexity, $CCPC$

This factor indicates to what extent the class paths act on class attributes. In fact, as in method-based object-oriented languages, several class methods can work on the same attributes; thus, in TROL, several paths can work on the same attributes. For this reason, the complexity of path coupling in working on attributes can be estimated by using:

$$CCPC = \sum_{i}^{NA} \sum_{j}^{NP} AC_i \; PRS(i,j)$$

where $PRS(i,j)$ is the number of times attribute i is present in path j; AC is the already defined attribute complexity. Note that $CCPC$ is higher when the same attribute is frequently used in more than one path. Since in TROL (as in OSDL) the concept of Path (i.e., process in OSDL) is present only for basic object classes (basic block in OSDL), for non-basic object classes $NP = 0$ and, thus, $CCPC$ is equal to zero.

Class Internal Task Complexity, $CITC$

As to concurrency, the class internal complexity mainly depends on the number of tasks which are present inside the class itself. For this reason, the $CITC$ has been considered equal to the number of internal processes of the class. If the class is a basic object class, the number of internal processes of a class is equal to the number of paths – i.e., $CITC = NP$. On the contrary, at the level of non-basic object classes, when the system under specification is only partially defined, the number of class paths is not yet defined, since the low-level classes are still undefined; in that case, it is supposed that the number of tasks is at least equal to the number of class sub-objects (i.e., attributes), thus $CITC = NA$.

Class Internal Communication Complexity, $CICC$

The measure of Class Internal Communication Complexity has a meaning only for non-basic object classes, while for basic object classes it is equal to zero, $CICC = 0$. Therefore, it is a complementary measure

with respect to the already presented Class Path Complexity (CPC) where the relationships among paths and attributes are measured.
The structure of non-basic object classes is defined in terms of communicating sub-objects:

$$CICC = \sum_{i}^{NCON} MSGC_i \; CTYPE(i)$$

where: $NCON$ is the number of connections among the sub-objects of the class (see App.A.2); $MSGC_i$ is the message complexity that depends on the complexity of the message class; $CTYPE(i)$ is a coefficient which takes into account the kind of communication between objects. In TROL (see App.A.2), four types of communication mechanisms are obtained by combining different service types; hence, $CTYPE$ is equal to 4 for normal-to-normal, 3 for normal-to-buffered, 2 for available-to-buffered, and 1 for available-to-normal communications. The complexity of a message, $MSGC$, depends on the complexity of the class which has been used to define the message structure, since in the object-oriented approach messages are also class instances (i.e., objects), $MSGC$ is equal to the CC of the message. With this assumption the costs for manipulating the message itself are also included in that measure.

It should be noted that a value for $CICC$ can be obtained since the early stages of system specification. Therefore, this can be very useful as a basis for defining process-oriented metrics.

2.1.3 External Class Complexity, ECC

The External Class Complexity (ECC) takes into account the complexity of provided and required services (Provided Service Interface Complexity, $PSIC$, and Required Service Interface Complexity, $RSIC$, respectively) of the class, considering also their interdependencies by means of the measure of Service Complexity, SC. The SC term also measures to what extent the class is capable of producing autonomous requests for the outer objects:

$$ECC = w_{PSIC} PSIC + w_{RSIC} RSIC + w_{SC} SC$$

In the following, the terms $PSIC$, $RSIC$ and SC are discussed separately. The values of weights w_{PSIC}, w_{RSIC} and w_{SC} depend on the application field under analysis. In the case of real-time systems specification, they are usually assumed to be 1, 1, and 1.5, respectively. This choice is due to the fact that in real-time systems the dependency among objects through services is an index of the event propagation inside the system. Therefore, SC must have a greater influence on ECC with respect to the others.

Provided Service Interface Complexity, $PSIC$

The complexity of the provided interface of a class is estimated with:

$$PSIC = \sum_{i}^{NPS} MSGC_i \; PMEC_i \; PTCPS_i$$

where NPS is the number of provided services; $MSGC$ is the already defined message complexity (equal to the CC of the message); $PMEC$ is a weight which takes into account the type of communication mechanism of a provided service (equal to 1 if the provided service is **buffered** or 2 if it is normal); $PTCPS_i$ marks the presence of a temporal constraint for the provided service i (it is equal to 2 if the temporal constraint specifying the service rate has been defined, otherwise a value equal to 1 is assigned).

Required Service Interface Complexity, $RSIC$

Differently from the method-based object-oriented models, in which the requests of services are hidden inside the methods body, in the case of the IO-based model (such as in TROL or OSDL languages), the complexity of the required interface of a class can be simply estimated by means of:

$$RSIC = \sum_{i}^{NRS} MSGC_i \; RMEC_i \; PTCRS_i$$

where NRS is the number of required services; $MSGC$ is the already defined message complexity; $RMEC$ takes into account the type of communication mechanism of a required service (equal to 1, if the required service is **available**, or 2 if it is normal); $PTCRS_i$ marks the presence of the temporal constraint for the required service i (it is equal to 2 if the temporal constraint specifying the service rate has been defined, otherwise a value equal to 1 is assigned).

Service Complexity, SC

The complexity of class services, SC, indicates to what extent the required services (outputs) of a class depend on the provided services of the same class. Therefore, it can be considered as a measure of the class behavior complexity. In general, for a class, two kinds of required services can be present – i.e., autonomous and non-autonomous. The former are those which are generated directly from the class without receiving any external request. The latter are those which are indirectly due to requests of other system objects. For these reasons, the Service Complexity is defined as:

$$SC = NAR + NASC$$

where NAR is the Number of Autonomous Requests. This is equal to the number of required services which are generated by paths independently of the provided services of the class. On the contrary, the Number of Non-Autonomous Requests of a class, $NNAR$, is equal to $(NRS - NAR)$. Hence, the complexity of non-autonomous services of a class, $NASC$ is estimated by:

$$NASC = \sum_{i}^{NNAR} NPS(Path(i))$$

where $NPS(Path(i))$ is the number of provided services which are present in the $Path(i)$, and function $Path(i)$ identifies the path which generates the non-autonomous request i.

The measure of SC can be directly obtained by measuring the definition of class clauses since the early phases of the software life-cycle.

2.1.4 Class Size, CS

The Class Size (CS) is the weighted sum of the Internal and the External Class Size (ICS and ECS):

$$CS = w_{ICS} \; ICS + w_{ECS} \; ECS$$

where w_{ICS}, and w_{ECS} are tailorable weights. Note that, since ICS and ECS take into account the size of the class by considering all its superclasses, then CS also includes such measure.

The class size can be very useful to evaluate the productivity and to predict the dimensions of the system under analysis and, thus, the effort of development, since the ECS can be estimated even if the system (i.e., the class) is only defined at its external level.

2.1.5 Internal Class Size, ICS

The Internal Class Size (ICS) takes into account the Class Path Size (CPS) and the Class Attribute Size (CAS) according to the TROL and OSDL models. The size of the class also depends on the communications among its sub-objects; this fact is measured with the Class Internal Communication Size, $CICS$. Therefore, the internal class size is defined as:

$$ICS = w_{CAS}\, CAS + w_{CPS}\, CPS + w_{CICS}\, CICS$$

where w_{CAS}, w_{CPS}, and w_{CICS} are application-dependent weights. In the following, the above terms are discussed.

Class Attribute Size, CAS

The size of each attribute (Attribute Size, AS) is equal to the class size of the attribute itself, thus $AS = CS$. Hence, the class attribute size, CAS, is obtained by considering the class attributes:

$$CAS = \sum_{i}^{NA} AS_i$$

In many object-oriented languages, such as in C++, there is a set of predefined basic types which are not implemented as classes. For these basic types the class size must be predefined.

Class Path Size, CPS

The Class Path Size is estimated by using the expression:

$$CPS = \sum_{j}^{NP} PTS_j$$

where PTS_j is the size of path j:

$$PTS_j = NT_j + \sum_{i}^{NS_j} NIST_i$$

where: NS_j is the number of states of path j; NT_j is the number of conditions for transitions of path j; $NIST_i$ is the number of statements (assignment or procedure call) associated with the state i of path j.

In TROL, the concept of Path is present only for basic object classes, while for non-basic object classes $NP = 0$; therefore, the value of CPS is equal to zero.

Class Internal Communication Size, $CICS$

In the TROL model, the measure of Class Internal Communication Size has a meaning for non-basic object classes where it is equal to the number of communications (i.e., connections in TROL) among sub-objects (i.e., attributes); therefore, $CICS = NCON$. In basic object classes, the connections are missing, hence $CICS = 0$. A value for $CICS$ can be obtained since the early stages of system specification. Therefore, it can be very useful as a basis for defining a process-oriented metric.

2.1.6 External Class Size, ECS

The External Class Size (ECS) takes into account the size of the external class interface, by considering the number of clauses, and that of provided and required services:

$$ECS = NPS + NRS + NC$$

2.2 Cognitive Metrics

In the object-oriented approach, cognitive metrics should give a measure of how easy it is, for the user, to understand the nature of a class by observing its external description. To this end, a significative cognitive measure is given by the Complexity Ratio, CR. Moreover, the estimation of system and subsystem verifiability is another important cognitive metric, this factor being measured by the Verifiability Index, VI.

2.2.1 Complexity Ratio, CR

The Complexity Ratio indicates to what extent the nature of a given class can be understood through its external description. This measure is very useful to estimate the cost of reuse of a class by means of its understandability. In fact, during the reuse of a class the analyst usually does not read all the details of class implementation, and observes only the class description. Therefore, CR is defined as:

$$CR = \frac{ECD}{ICC}$$

where ECD is a measure of the External Class Description, and ICC is the Internal Class Complexity. A high CR means that the class external description is very detailed with respect to its internal complexity; therefore, much of the whole class behavior can be understood by observing the external class description. It has been necessary to define ECD instead of using the already defined ECC, since the latter does not represent only the external class complexity as can be observed by the external class description. In fact, ECC has also been defined by considering the SC, which in turn is based on the knowledge of dependencies between required services and class paths. The ECD gives a measure of the external class description as can be observed by the analyst in the phase of class reuse. In method-based models, the external class description consists in the description of methods domain plus their preconditions and postconditions, if these are available at the external class level. In other languages, assertions or clauses for describing the internal class behavior at the external level can be present. In the this case, the External Class Description is more complete as in TROL. Note that, in TROL, the external class description includes provided and required services measuring the class static interface, augmented by clauses measuring the class behavior.

ECD is defined as the weighted sum of Class Static Description, CSD, and Class Dynamic Description, CDD. The static description covers the structural aspect, and the dynamic description the behavioral aspect:

$$ECD = w_{CSD}\, CSD + w_{CDD}\, CDD$$

where w_{CSD} and w_{CDD} are application-dependent weights – for example, in measuring real-time system specifications, values 1 and 2, respectively, have been usually assumed in order to give more importance to the description of system behavior.

CSD is obtained by means of the $PSIC$ and $RSIC$ already defined in Sect.2.1.3:

$$CSD = w_{PSIC}\ PSIC + w_{RSIC}\ RSIC$$

CDD describes how many details regarding the class behavior are reported in the class external description. For the TROL model, this measure holds:

$$CDD = \frac{NSPSR}{NPS} + \frac{NSRSR}{NRS} + CCC$$

where $NSPSR$ is the number of specified **Provided_service** rates; $NSRSR$ the number of specified **Required_service** rates; and CCC is the Class Clause Complexity defined as:

$$CCC = \sum_{i}^{NC} CLC_i$$

where CLC is the CLause Complexity estimated with:

$$CLC_i = NPS_i + NRS_i + NOP_i + PTCC_i$$

while NOP_i is the number of Boolean and comparative operations in clause i; NPS_i is the number of provided services in clause i; NRS_i is the number of required services in clause i; $PTCC_i$ marks the presence of temporal constraint for clause i (it is equal to 1 if the temporal constraint has been defined, otherwise it receives a value equal to 0).

2.2.2 Verifiability Index, VI

In TROL, class and system verifiability is guaranteed by means of the definitions of provided services, required services, and clauses with their respective temporal constraints. Basic object classes, such as XSMs and XCMs, are implemented according to the clauses defined in their external class interface. Therefore, the external class description given in terms of clauses must verify the class implementation expressed by means of a set of communicating sub-objects (i.e., non-basic object class) or as a basic object class (i.e., XSMs, XCMs). To this end, the verifiability index for a given class is defined by the ratio between the ECD and CC:

$$VI = \frac{ECD}{CC}$$

This index indicates to what extent the class can be verified by using the information contained into the external class description with respect to its actual complexity. For example, if a given class reports to the external level (by means of provided services, required services and clauses) a lot of details with respect to its total complexity, then it is more verifiable than a class that, having the same total complexity, is less described in detail at the external level. Therefore, VI can be regarded as measure of class verifiability.

In TROL, this measure can also be used to evaluate the testability of the system under development. In fact, in the TROL language the patterns for testing the application can be generated by using the external class description with services and clauses.

2.3 Process-Oriented Metrics

When Managers and Software Quality Assurancers plan budgetary and personnel resources for a new project, they feel more comfortable to use estimation models than just to rely on rules of thumb or entirely subjective judgments. For this reason, they need objective information from the projects under development. At the process level, the metrics which can be useful to control the development are productivity and expected cost [Henderson-Sellers et al. 1993]. These measures are usually based on the trend of technical metrics, which in turn, to be suitable, must be capable of producing significative results at each level of system development. In fact, according to the TOOMS/TROL model congruent values for the technical metrics can be estimated since the early stages of the software life-cycle. In the following, coefficients to estimate team productivity, system reuse level and expected cost, are reported. Another very important factor is the evaluation of the need for testing during the development process. Though the phases of system testing should be defined in advance, it is also important to detect when a class or a sub-system under development has grown too much without being tested. These conditions can be easily detected by using the last metric of this section.

2.3.1 Productivity Index, *PI*

The measurements of productivity are very complex for object-oriented systems since in these cases these cannot be reduced to a simple ratio between the quantity of developed software and the effort of development. In fact, a good measure of productivity should be defined as the ratio between the increase in implemented system functionalities and the effort to add them. In effect, in the case of object-oriented languages and methodologies, productivity should often result higher than that obtained in a classical procedural development environment, since the mechanisms of specialization and aggregation are powerful tools for software reuse and thus for increasing productivity. This fact must be considered in estimating productivity. Therefore, at each time instant of system development, a measure of productivity can be given by a Productivity Index, PI, defined as:

$$PI = \sum_{i}^{NewCls.} CS_i + \sum_{i}^{NewSpec.Cls.} PINCCS_i$$

where: $NewCls.$ are the classes which have been added (without any specialization) with respect to the last check point of PI estimation; $NewSpec.Cls.$ are the classes which have been specialized with respect to the last PI estimation. $PINCCS$ is the productivity in incrementing the size of class i:

$$PINCCS_i = CS_i - \sum_{j}^{SuperClassesOf(i)} CS_j \;\; CR_j$$

where $SuperClassesOf(i)$ are only the direct superclasses of class i. The increment of class size is obtained by considering the class size, CS_i, of class i, minus the sum of the size of super-classes of class i. This last term is obtained by weighting the class size of each super-class with its complexity ratio, CR; in this way, a higher productivity is assigned to less understandable classes, and a lower productivity to classes which are easily understandable.

Note that PI indicates to what extent the software has been produced by the designer with respect to the last PI estimation, and is obtained in terms of class size. Since this measure is obtained with respect to the last value, the sequence of PI should be estimated by using a constant interval in terms of working days. A version of this metrics, independent of the interval duration, can be obtained by dividing PI for the time interval expressed in days of work – i.e., $NPI = PI/DW$.

2.3.2 Reuse Index of the system, RI

The reuse index indicates to what extent the reuse mechanism has been exploited in the system specification. The most important and used mechanism of reuse of the object-oriented paradigm is the specialization. Therefore, a good measure of the reuse level in the system is given by the Reuse Index, RI, defined as:

$$RI = \frac{\sum_i^{AllSpec.Cls.} INCCS_i}{\sum_i^{NCL} CS_i}$$

where $INCCS_i$ is the increment of class size for the specialized class i; $AllSpec.Cls.$ are the classes which have been specialized with respect to the initial condition of class library. In this case, $INCCS$ is estimated by considering as superclasses only those which belong to the class library:

$$INCCS_i = CS_i - \sum_j^{SuperClassesOf(i)} CS_j$$

RI can be estimated at each time instant of system development giving in this way the trend of reusability.

2.3.3 Expected System Cost, $ESCost$

Most of the methods proposed in the literature express the effort as a function of a number of quantities which can be usually determined in the early phases of the software life-cycle or directly from system requirements. For example, according to the Albrecht's method, the effort estimation is bases on the system size which in turn is evaluated by using a set of features such as the number of internal and external files, the number of inputs and outputs, and the number of external enquiries, etc. Finally, the system size is multiplied by a factor depending on the technical complexity of the application. This factor is heuristically estimated by means of a set of questions which are used to tune the effort estimation with respect to the application domain context [Albrecht et al. 1983], [Rask et al. 1993]. Another example is the COCOMO model [Boehm 1984], [Boehm 1982], which estimates the effort on the basis of the system size in LOC and the use of 15 cost drivers. The values of these drivers are identified according to the development context – i.e., type of computer, qualification of personnel staffing, etc. Other techniques following the same path have been compared in [Kemerer 1987], [Laranjeira 1990]. The drivers usually adopted cannot be easily tuned for tailoring the method for measuring the specifications of real-time systems in formal languages. In fact, this fails to take into account the effort of defining class behavior when it is strongly dependent on temporal constraints. Therefore, different parameters must be taken into account to measure the effort for specifying real-time systems. In these cases, the effort is dependent on system complexity and specification requirements rather than on the number of inputs and outputs files that sometimes can be missing (for example, in embedded systems). In addition, as has been previously pointed out, in the case of object-oriented real-time specification languages, the measure of system size in LOC is not adequate to express the system size and, hence, the effort of development. This is due to the fact, that the size of the specification in dedicated languages is not directly correlated to the size of the final code (usually specification languages – e.g., OSDL, TROL, etc. – are translated in conventional languages such as C++, ADA, etc.).

For these reasons, the Expected System Cost, $ESCost$, can be regarded as the sum of costs which are due to the reuse of old classes, $ESCost_{reus.}$, plus the cost of development of new classes, $ESCost_{dev.}$, plus the cost of specialization of already implemented classes, $ESCost_{spec.}$:

$$ESCost = ESCost_{dev.} + ESCost_{reus.} + ESCost_{spec.}$$

The cost related to the development of new classes takes the form:

$$ESCost_{dev.} = CMA_{dev.} \sum_{i}^{NewCls.} CC_i$$

where $CMA_{dev.}$ is a company-dependent coefficient which expresses the mean cost of development per class in man months (estimated by considering the average costs observed in a congruent category of already developed applications); and $NewCls.$ are the classes which have been implemented without any specialization or reuse in general. The costs of reuse and specialization hold:

$$ESCost_{reus.} = CMA_{reus.} \sum_{i}^{ReusedCls.} \frac{CC_i}{CR_i}$$

$$ESCost_{spec.} = CMA_{spec.} \sum_{i}^{NewSpec.Cls.} INCCS_i$$

where: $CMA_{reus.}$ and $CMA_{spec.}$ are company-dependent coefficients, the first expresses the mean cost of reuse per class in man month, while the second is the cost of specialization in man months; $INCCS$ is the already defined increment of class size. $ESCost_{reus.}$ is defined on the basis of the ratio between CC and CR for each reused class. This ratio is lower when the class reuse is easier, since CR is bigger. The coefficients CMAs also depend on the language used such as the estimation of the Albrecht's Function Point [Albrecht et al. 1983], [Fenton 1991]. The definition of CMA values is statistically based; they allow to apply past experiences to project the future. Software process data may be gathered as a part of the every day process control activities.

It should be noted that, for the estimation of the general cost, both technical and cognitive metrics defined in the previous sections have been used. This is possible since most of those measures lead to a value even if the system is only partially specified. This characteristic is congruent with TOOMS/TROL, TRIO+, OSDL, etc. capabilities.

2.3.4 Test Index, TI

At each time instant of the developing process, it is very useful to identify the classes (i.e., sub-systems) for which the test is needed before continuing with the developing process. With this metric it is assumed that the test of a class is needed when its complexity has grown too much with respect to the last testing section. In TROL, the testing can correspond to (i) the verification of the high-level class behavior by means of clauses execution, (ii) the validation of system composition/decomposition and (iii) the validation of an XSM in terms of clauses [Nesi 1993]. Analogous mechanisms are present in similar languages.

Therefore, the Test Index for a given class i is estimated as a relative change in class complexity:

$$TI_i = \frac{CC_i(ActualVersion) - CC_i(LastTestedVersion)}{CC_i(LastTestedVersion)}$$

As a conclusion, classes which have a TI bigger than a predefined threshold must be tested before continuing with the development process. The threshold is defined on the basis of company experience in the specific field of the application measured. This measure can be extended to the entire system by evaluating the TI of the class representing the system itself.

3 Metric Validation

In general, metrics can be divided in *predictive* and *descriptive*. *Predictive* metrics try to predict the future value of a system characteristic by using simple and early measurements on the system itself. This type of metrics is very difficult to be validated since the entire process of system development must be controlled. Several projects must be monitored, and the results are only a picture of the company under control. *Descriptive* metrics try to estimate a given characteristic of the software system under metrication by means of *direct* or *indirect* measures (the aim of direct measure is usually to estimate values for predictive metrics and for measuring the product quality). In the case of *direct* measure, the metric obviously does not need to be validated; for example, the measure of the LOC for estimating the program length. It should be noted that direct estimations are usually complex and computationally heavy to be performed; for these reasons, they are usually not preferred. In the case of *indirect* estimations, the parameters measured must be related to the characteristic under measure (for example the measure of the number of classes can be supposed to be related to the system complexity by means of a mathematical relationship). In this case, the process of metric validation consists in (i) evaluating the mathematical relationships between the parameters measured and the real value of the system characteristic under evaluation (i.e., the mathematical model), and (ii) verifying the robustness of the identified relationship. This relationship can be linear or not, and it must be identified by using both mathematical and statistical techniques – e.g., [Stetter 1984], [Fenton 1991], [Zuse 1994].

The above-mentioned metrics have been defined, according to the evaluation process model of ISO 9126. In the model proposed, there are many descriptive metrics, such as CC, CS, CR, VI, PI, RI, TI; among these, a discussion about the validation of CC, CS, VI, and CR is presented. These have been used for measuring a set of test projects in order to validate them. Therefore, it will be demonstrated that: (i) the class complexity (CC) and the real class complexity (RCC) are linearly correlated (correlation of Pearson [Fenton 1991]), and (ii) the class size (CS) and the real class size (RCS) are linearly correlated. Since most of the other cognitive and process-oriented metrics proposed depend on the above-mentioned technical metrics, once these are validated, PI, RI, and TI can be considered *direct descriptive metrics*.

The metric validation has been performed with respect to real data. The data comes from the University of Florence (Italy) and O.T.E. S.p.a., a company which manufactures micro-telephones and other real-time embedded systems. The projects are a significative set of software for embedded systems: (i) the buoy system proposed by Booch [Booch 1986], and revised by [Sanden 1989a]; (ii) an elevator system with two columns and six floors proposed in the literature – e.g., [Sanden 1989b], [Gangopadhyay et al. 1993]; (iii) and a filter for digital signals, (iv) to (vi) the control software for different hand-held wire-less micro-telephones. In Tab.1, the main characteristics of this data are reported. It should be noted that, languages such as OSDL, TROL need to be translated in a standard language (e.g., ADA, C++) to be compiled and

Project	NCL	$NNBOC$	$NBOC$	$NProc.$	$NInst.$	$Eff.$	Lev
buoy system	10	3	7	14	295	5	exp.
elevator	10	2	8	18	433	6	exp.
filter	4	1	3	6	168	2	exp.
microtel.1	18	6	12	22	570	20	beg.
microtel.2	35	6	29	49	1352	26	beg.
microtel.3	31	6	25	42	1050	35	beg.

Table 1: Summary of the main characteristics of the data adopted for metrics validation where: NCL is the number of classes of the system, $NNBOC$ the number of non-basic object classes in the system, $NBOC$ the number of basic object classes in the system, $NProc.$ the number of concurrent processes which are present in the application, $NInst.$ the number of TROL instructions of control, $Eff.$ the effort in working days for *specifying* the system under development (hence, without considering the costs of analysis), $Lev.$ the experience of developers in using TROL (*exp.* for experts and *beg.* for beginners).

executed at run-time. In particular, once a system specified in TROL is translated in C++, it has to be compiled and linked with other 70 classes approximately of the TROL kernel. Many of these classes are superclasses of those obtained from translating the TROL classes in C++ [Bucci et al. 1993]. Therefore, the projects adopted represent real problems and not simple textbook examples.

In Tab.2 and Tab.3, the correlations among the most important metrics previously proposed are reported. Since the measurement values are strongly different in the cases of basic and non-basic object classes according to TROL, OSDL (blocks and processes, respectively) languages, two different tables of correlations have been used for basic and non-basic object classes. The values for the metrics defined have been estimated by assuming all the weight values to be equal to 1.0 except for w_{SC} and w_{CDD} which have been posed to be equal to 1.5 and 2.0, respectively (for the previously discussed reasons). The real class complexity, RCC, has been estimated by evaluating the state domain of the class or sub-system under estimation, while the real class size, RCS, has been measured by counting the LOC in the run-time version of the project (obviously neglecting the TROL kernel and support library).

As can be seen from the correlation matrices the metrics proposed are only partially correlated, while metrics for estimating class complexity and size are strongly correlated with the respective real values. It can also be observed that the class complexity, CS, cannot be considered as a good measure of the real class complexity, RCC, as usually assumed for traditional languages. This is confirmed by the correlations between RCC and RCS for both basic and non-basic object classes. In fact, in both cases, the correlation between CC and RCC is bigger than that of CS with RCC. This is particularly true in the case of basic object classes, where the details of class behavior are dominant with respect to the structural aspects considered in the ECC. In fact, the best estimation of class complexity is obtained by using the internal class complexity, ICC. In Fig.4 the diagrams of the relationships between the class complexity and class size with respect to their respective real values are reported.

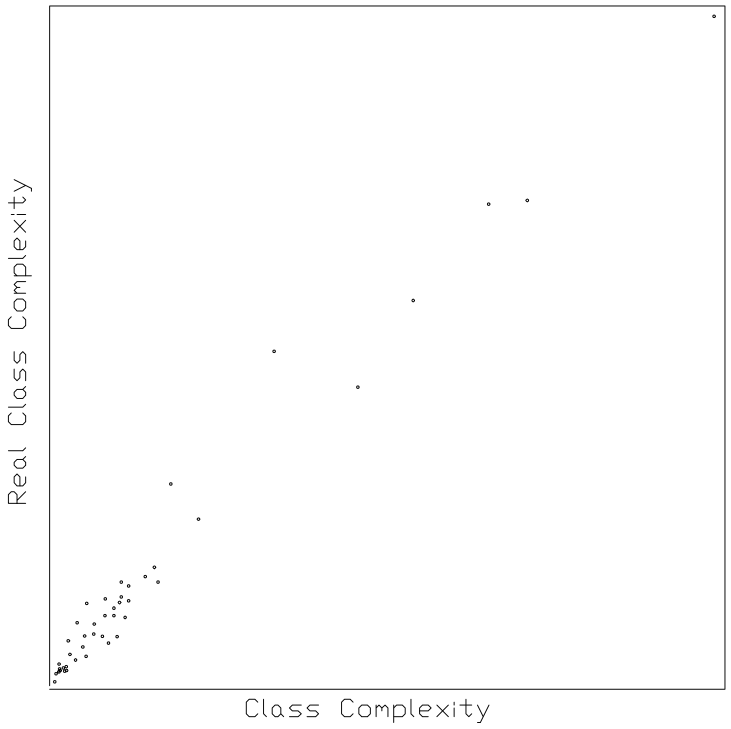
As can be noted from the correlation matrices the ECC and ECS are quite correlated with RCC and RCS, respectively. This fact is very useful for estimating the class complexity and size even if the class is only partially defined. This means that the estimation of system complexity and size can be obtained even if the system is only partially specified (see Fig.2). For example, in case (b) of Fig.2 the CC of class A is estimated by considering the ECC of class B, the CC of class C and the CC of class D, which in turn

	non-basic object classes										
	CC	ICC	ECC	CS	ICS	ECS	CR	ECD	VI	RCC	RCS
CC	1.000										
ICC	0.999	1.000									
ECC	0.974	0.972	1.000								
CS	0.989	0.989	0.960	1.000							
ICS	0.981	0.989	0.959	0.999	1.000						
ECS	0.972	0.972	0.964	0.981	0.980	1.000					
CR	-0.755	-0.754	-0.760	-0.727	-0.730	-0.647	1.000				
ECD	0.767	0.763	0.836	0.704	0.700	0.795	-0.128	1.000			
VI	-0.744	-0.743	-0.746	-0.719	-0.722	-0.636	**0.985**	-0.401	1.000		
RCC	**0.982**	0.971	0.955	0.904	0.997	0.972	-0.732	0.700	-0.723	1.000	
RCS	0.880	0.991	0.963	**0.998**	0.998	0.983	-0.746	0.693	-0.739	0.983	1.000

Table 2: Correlations among the technical and cognitive metrics proposed for non-basic object classes with respect to the actual values of complexity and size (RCC and RCS, respectively).

	basic object classes										
	CC	ICC	ECC	CS	ICS	ECS	CR	ECD	VI	RCC	RCS
CC	1.000										
ICC	0.939	1.000									
ECC	0.729	0.449	1.000								
CS	0.714	0.614	0.631	1.000							
ICS	0.776	0.714	0.595	0.961	1.000						
ECS	0.301	0.145	0.492	0.734	0.252	1.000					
CR	-0.339	-0.544	0.200	-0.093	-0.281	0.406	1.000				
ECD	0.553	0.325	0.790	0.814	0.668	0.880	0.429	1.000			
VI	-0.314	-0.485	0.148	0.095	-0.141	0.465	**0.902**	0.542	1.000		
RCC	**0.864**	0.771	0.771	0.700	0.801	0.197	-0.292	0.428	-0.292	1.000	
RCS	0.632	0.601	0.447	**0.934**	0.877	0.713	-0.104	0.724	0.111	0.713	1.000

Table 3: Correlations among the technical and cognitive metrics proposed for basic object classes with respect to the actual values of complexity and size (RCC and RCS, respectively).



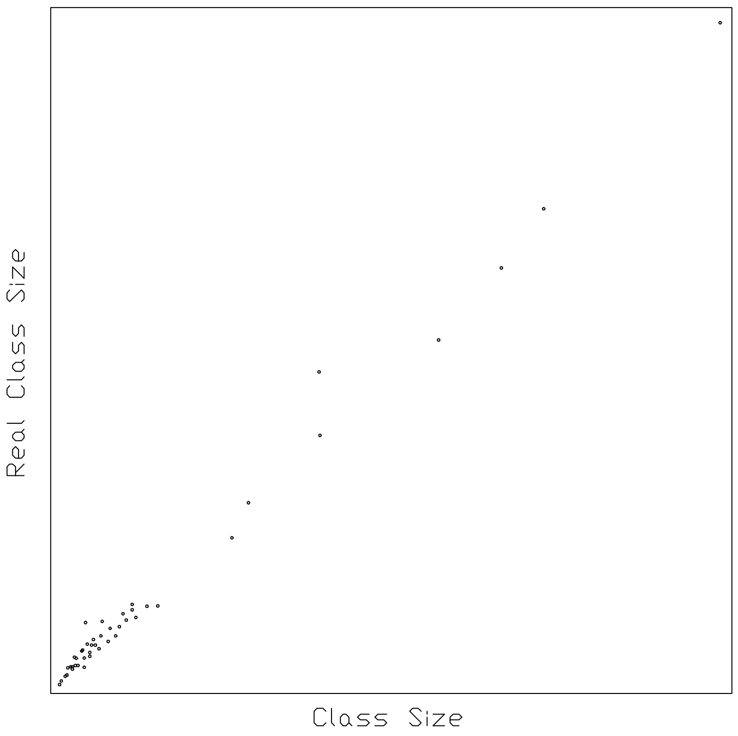


Figure 4: Relationships between (a) the class complexity and the real class complexity, (b) the class size and the real class size.

is estimated by considering the ECCs of classes G and H. When a class complexity is estimated by using only a part of the information (e.g., CC estimated by using only ECC since the ICC is not yet available) the weights in the expression of CC can be modified in order to compensate the loss. In these cases, the values of weights are estimated by using the mean values of the metrics and, thus, the previous experiences (our values are reported in the following). For example, in the case of a basic object, if the ICC is missing w_{ECC} is assumed equal to 3. For these reasons, the model proposed allows the metrication at each level of specification detail, and thus the system development can be continuously maintained under metrication.

The validation of complexity ratio, CR, to demonstrate its efficiency in measuring to what extent a class can be understood by observing its external class description is quite unfeasible because the measure of understandability is subjective. In addition, it should be noted that, CR is strongly correlated with the verifiability index, VI (see Tab.2 and Tab.3). Our experiments have shown that the verifiability of basic object classes in the sense of formal verification of class behavior [Bucci et al. 1994] is guaranteed when $VI > 1$. For non-basic object classes, VI depends on CC which tends to grow with the system size; therefore, to get a measure of VI, independent of the system size, VI should be obtained by dividing ECD by a normalized version of CC with respect to the number of class attributes. For these reasons, VI can be used for measuring both understandability and verifiability.

4 General Application of Metric Framework

The integration of the metric framework into the specification tool allows to save the time of measurements since watchdogs on the values of certain metrics can be set in order to maintain the quality at the imposed values. This is also a way to compel the developers to produce specifications with a predefined quality. The quality model proposed has been implemented according to the evaluation process model of ISO 9126 and the GQM approach [Basili et al. 1984]. Therefore, *quality requirement definition*, *evaluation preparation* and *evaluation procedure*, have been the steps accomplished to reach the results presented in this paper. According to ISO 9126, quality metrics has been selected and for these the rating values are defined; hence, the validation of metrics with respect to real measures has been proposed. Thus, the model proposed consists in a set of metrics capable of expressing meaningful indicators and some conformance to the ISO 9126 standard.

In the next subsections, four aspects will be discussed: (i) the selection of metrics defined for measuring general concepts, (ii) a discussion about the language dependence of the metrics proposed, (iii) the quality guidelines based on the metric framework proposed, and (iv) a discussion about the adoption of the metric proposed as a support for object-oriented methodologies.

4.1 Classification scheme

The classification scheme reported in Tab.4 explains some aspects of the metric framework integrated in the TOOMS tool. This Table shows the mapping of the metrics proposed into the general concepts of system measurements. As was pointed out by many authors, metrics must fit the main needs of developers and managers; data collection and data gathering have to be an automatic support for the development process. In fact, the classification is related to a real need for gathering different kinds of metrics and for inducing metrics to produce a readable and understandable data output.

In Tab.4 four levels are given, that is, *class-level*, *test-level*, *reuse-level*, and *process-oriented level.* In the metrics framework proposed, a correlation exists among the different levels and metrics. In fact, some metrics assume different meanings according to the level at which they are adopted. Thus, the same metrics can be used by different levels for analyzing distinct aspects. The first two levels are very useful for estimating specific properties of the software produced. For each class, the features size and complexity can be estimated. In particular, a strong emphasis is given to the internal concurrency of the class and to the possibility of taking into account the presence of temporal constraints, making this model very suitable for analyzing object-oriented formal languages for real-time systems. Moreover, by using the metrics of *test-level* the testability and verifiability of each class as well as of the whole system can be estimated. The metrics classified into the *reuse-level* can indicate to what extent the available class libraries have been exploited and what is the cost of reuse for a given class. For example, CR is used during the class reuse for measuring the effort needed to understand the class nature and, thus, to reuse it; while RI is very useful to evaluate the level of reuse in a given application. This category of metrics plus those of the *process-oriented level* can be employed to keep under control the quality of software development. In TOOMS, this is possible because the user can evaluate in each instant of the system specification several metrics, even if the system is only partially specified.

4.2 Language dependency

The metrics proposed can be classified as high- and low-level metrics depending on their relationships with the other metrics of the framework (see Tab.5[1]), where:

- *Lev.* is the metric level (H for high-level metrics, L for low-level metrics, M are parameters obtained by a direct measure, V are coefficients with an assigned value.). The low-level metrics are mainly based on the direct measures, while the high-level metrics are defined on the basis of the low-level metrics.

- *L.Dep.* is a vote from 0 to 4 expressing to what extent the definition of the metric depends on the language adopted, where 0 means that the metric is independent of the language used for implementing the system under metrication, while for higher values an increasing dependence is present. As can be observed, most of the high-level metrics are language independent. For these reasons, the tailoring of the metrics framework for a language other than TROL consists only in redefining some of the low-level metrics.

- *Type* is the main classification of the paper and reports the prominent nature of the metrics proposed: T for technical, C for cognitive, and P for process-oriented metrics. As can be noted, the process-oriented metrics are completely language independent, while the cognitive metrics are lightly dependent and the technical metrics are the most dependent on the language. Measures having *Lev.* equal to M or V have not been classified since they are simple direct measures or constants, though some of them, such as the number of class attributes, could be considered technical or cognitive metrics.

[1]In the same table the direct measures which can be estimated by the observation of elementary measurable characteristics are also reported.

General Concepts	Measurements
Class-Level	
Class External Interface	ECC, ECD
Complexity	ECC, $PSIC$, $RSIC$, SC
Expressivity	ECD, CSD, CDD
Class Internal Features (attributes, operations, etc.)	CAC, CPC
Class size	CS, ECS, ICS
Attributes	AS, CAS
Methods	PTS
Class complexity	CC, ECC, ICC, VI
Attributes	AC, CAC
Methods (logic complexity, paths or processes)	CPC, PIC, $CITC$
Method coupling, cohesion, etc.	$CCPC$
Timed complexity (temporal constraint on services and clauses)	$PSIC$, $RSIC$, CCC
Input/Output	NRS, NPS, PEC
Communication	$CICC$, $RSIC$, PEC, $PSIC$
Test-Level	
Complexity Metrics	CR, ECD, CC
Timed complexity (temporal constraint on services and clauses)	CCC, $PSIC$, $RSIC$
Logic complexity (class paths)	ECD, $CCPC$, $CITC$
Test coverage	VI, TI
Reuse-Level	
Inheritance	RI
Class Features (attributes, services, etc.)	CR, ECD, ECC
System/Subsystem	ECC, ECD, $CICC$
Cognitive Metrics	CR, VI
Process-Oriented Level	
Productivity	PI
Reuse	RI
Size estimation	CS, ICS
Complexity estimation	CC, ICC, ECC
Cost estimation	$ESCost$

Table 4: Mapping of general concepts towards TOOMS metrics.

Metric	Lev.	L.Dep.	Type	Note
AC	H	0	T	Attribute Complexity, CC of attribute;
AS	H	0	T	Attribute Size, CS of attribute;
CAC	H	0	T	Class Attribute Complexity;
CAS	H	0	T	Class Attribute Size;
CC	H	0	T	Class Complexity;
CCC	H	2	C	Class Clause Complexity;
$CCPC$	H	0	T	Class Coupling Path (or process) Complexity;
CDD	H	2	C	Class Dynamic Description;
$CICC$	H	0	T	Class Internal Communication Complexity;
$CICS$	H	0	T	Class Internal Communication Size;
$CITC$	H	0	T	Class Internal Task Complexity;
CLC	L	3	C	CLause Complexity;
CMA	V	2	-	A company-dependent coefficient;
CPC	L	0	T	Class Path (internal process) Complexity;
CPS	L	0	T	Class Path (or process) Size;
CR	H	0	C	Complexity Ratio;
CS	H	0	T	Class Size;
CSD	H	1	C	Class Static Description;
$CTYPE(i)$	V	2	-	Communication Type of connection i;
DW	M	0	-	Days of work;
ECC	H	0	T	External Class Complexity;
ECD	H	0	C	Measure of the External Class Description;
ECS	H	1	T	External Class Size;
$ESCost$	H	0	P	Expected System Cost;
ICC	H	0	T	Internal Class Complexity;
ICS	H	0	T	Internal Class Size;
$INCCS$	H	0	P	Increment of Class Size;
$MSGC$	L	2	T	Message Complexity;
NA	M	0	-	Number of class Attributes;
NAR	M	0	-	Number of Autonomous Requests in the class;
$NASC$	L	0	T	Complexity of Non-Autonomous Services of a Class;
NC	M	2	-	Number of Clauses in the class;
NCL	M	0	-	Number of system Classes;
$NCON$	M	0	-	Number of Connections in the class;
$NIST$	M	0	-	Number of statements;
$NNAR$	L	0	T	Number of Non-Autonomous Required services in the class;
NOP	M	2	-	Number of Boolean and comparative operations;
NP	M	0	-	Number of class Paths;
NPI	H	0	P	Normalized PI;
NPS	M	0	-	Number of Provided Services;
$NPS(j)$	M	1	-	Number of Provided Services used in Path j;
NRS	M	2	-	Number of Required Services;
$NSPSR$	M	3	-	Number of Specified Provided Services Rates;
$NSRSR$	M	3	-	Number of Specified Required Services Rates;
NT	M	2	-	Number of conditions for Transition in a Path or process;
$Path(i)$	M	2	-	Path generating the required service i;
PEC	L	1	T	Path (or process) External Complexity;
PI	H	0	P	Productivity Index;
PIC	H	0	T	Path (or process) Internal Complexity;
$PINCCS$	H	0	P	Productivity Increment of Class Size;
PMC	L	1	T	McCabe-like Complexity of statements in a Path;
$PMEC$	L	2	T	Weight of communication Mechanism of Provided services;
$PRS(i,j)$	M	1	-	Number of Presences of attribute i in path j;
$PSIC$	L	2	T	Provided Service Interface Complexity;
PTC	H	1	T	Class Path (or process) Complexity;
$PTCC$	M	4	-	Presence of Temporal Constraint for the Clause;
$PTCPS$	M	4	-	Presence of Temporal Constraint for the Provided Service;
$PTCRS$	M	4	-	Presence of Temporal Constraint for the Required Service;
PTS	H	1	T	Class Path (or process) Size;
RI	H	0	P	Reuse Index of a system or a sub-system;
$RSIC$	L	2	T	Required Service Interface Complexity;
$RMEC$	L	2	T	Weight of communication mechanims of Required service;
SC	H	0	T	Service Complexity of a class;
TI	H	0	P	Test Index of a class;
VI	H	0	C	Verifiability Index of a class;

Table 5: Classification of the metrics proposed.

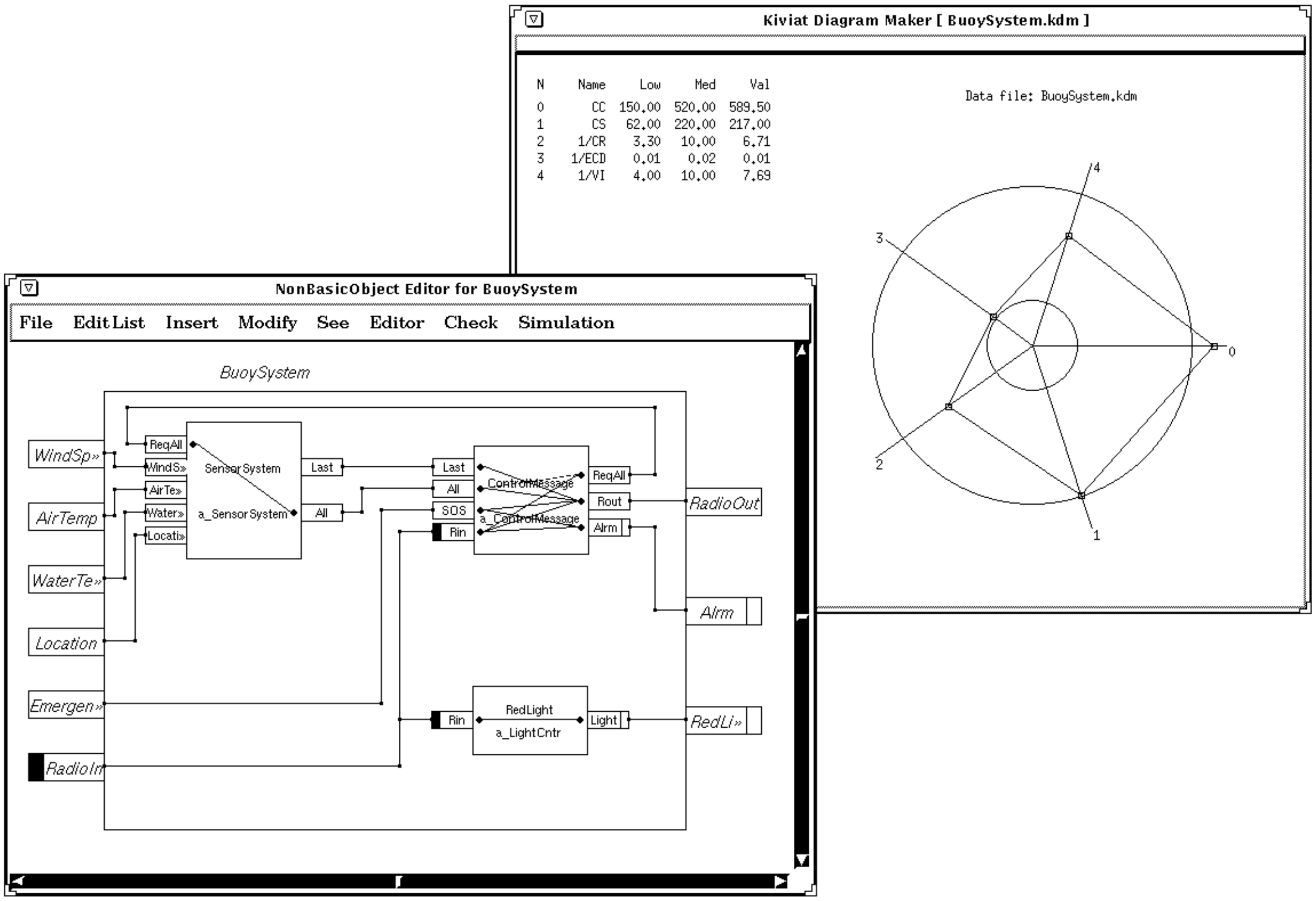
4.3 Quality Guidelines

According to the evaluation process model of ISO 9126, a set of metrics has been selected on the basis of their correlation with the respective characteristics of the software product. Initially, the correlations have been supposed, then their efficiency has been confirmed by experiments. On the other hand, since the metrics defined are for formal specification languages, it is obviously that it is highly probable that to a higher quality corresponds a lower complexity and size, since in those conditions a class can be more easily verified and validated even by using exhaustive testing. Therefore, the following guidelines have been defined:

- For each basic object class, the fulfilling of constraints: $CC < Tcc$ and $CS < Tcs$ guarantees a selected software quality in terms of meeting maintainability and reliability constraints. The concept of using the class complexity as a measure of maintainability is not new as can be seen in [Li et al. 1993].
- For each basic object class, the fulfilling of constraint: $CR > Tcr$ or $VI > Tvi$ guarantees the class or system verifiability and understandability; thus, they will be more confident in meeting the system requirements and more reusable. If these guidelines have been satisfied for all basic object classes, then the non-basic object classes which are obtained by composition will maintain the same characteristic of verifiability and understandability if the ECD is close to a mean value (see Tab.6). These measures can be regarded as normalized measures of class quality.
- For each class the constraint $TI < Tti$ must be satisfied during the whole development process to guarantee the testability of newly added features with low effort and to avoid the degeneration which often leads to test too many new features at the same time. The evolution of TI can be very useful to control if the system has been maintained under test.

In Tab.6 the typical ranges of the main metrics proposed are reported. The values reported in this table have been used for defining the rating of our quality criteria in accordance with ISO 9126. In particular, for basic object classes Tcc, and Tcs have been imposed equal to the high values reported in Tab.6; while for the case of non-basic object classes Tcc, and Tcs have been imposed equal to the mean values. This is due to the fact that, in this case, the high values are only indicative of the maximum complexity and size of a system or subsystem. Therefore, for non-basic object classes VI and CR are more indicative measures of system/subsystem quality. For both basic and non-basic object classes Tcr and Tvi have been imposed to be equal to the mean values. In Fig.5, the Kiviat diagram (directly produced by the TOOMS tool) presents the values of CC, CS, $1/CR$, $1/ECD$, and $1/VI$ for the Buoy System previously discussed. The external circle reports the values of thresholds at the intersection with the respective axes; and the rays of the circle have been normalized. From the Kiviat's diagram, it can be seen that class complexity is too high while the other measures are quite good. In particular, being the BuoySystem a non-basic object class the $1/CR$ is the most indicative measure of class quality. In fact, it shows that the class is well specified at the external level with respect to its complexity, thus its behavior can be easily verified (see [Bucci et al. 1994] for the process of verification).

As regards the application of metrics, the model of metrics proposed is general enough to be used with other object-oriented approaches with minor changes – e.g., with [Mandrioli 1993], [Coleman et al. 1992], [NorthernTelecom 1993], [Braek et al. 1993]. Due to the presence of many differences among projects of



```
N     Name    Low     Med     Val
0       CC  150.00  520.00  589.50
1       CS   62.00  220.00  217.00
2     1/CR    3.30   10.00    6.71
3    1/ECD    0.01    0.02    0.01
4     1/VI    4.00   10.00    7.69
```

Figure 5: The TOOMS tool during the monitoring of BuoySystem class quality.

	basic object classes			*non-basic object classes*		
measure	low	mean	high	low	mean	high
CC	15.0	60.0	140.0	150.0	520.0	2000.0
ICC	8.0	42.0	100.0	140.0	450.0	1800.0
ECC	6.0	20.0	60.0	15.0	35.0	65.0
CS	8.0	30.0	60.0	62.0	220.0	570.0
ICS	4.0	20.0	45.0	55.0	200.0	540.0
ECS	4.0	9.0	20.0	8.0	14.0	27.0
CR	0.4	1.5	4.5	0.05	0.1	0.3
ECD	13.0	45.0	110.0	26.0	50.0	82.0
VI	0.3	0.8	2.2	0.06	0.1	0.25

Table 6: Typical values for the most important metrics proposed, for both basic and non-basic object classes.

the same type in different companies as well as in the same company, it is important to have the possibility of performing project-oriented tailored measures. To this end, in the metric framework proposed, there exists weights, and coefficients. These have been included for adjusting the metrics framework to specific type of project on the basis of the company experiences. In addition, weights and coefficients can also be modified for (i) better meeting the company goals – e.g., improve quality, improve reusability, etc., (ii) evidencing and measuring specific features – e.g., behavior, communications, structure, etc., and/or (iii) for compensating the presence of different languages and thus using a unique metric framework in non-homogeneous systems [Henderson-Sellers et al. 1994]. Therefore, developers which intend to adopt the metrics framework proposed in this paper have to measure at first a set of their projects by using our weights (in this way, a mean value for each metrics estimated on their projects is obtained), and then they can impose their weights and thresholds according to the company goals.

For most of the process-oriented metrics (e.g., PI, RI) developers do not need to get an absolute value, but they need to understand the trends of values in order to identify (i) discontinuity and changes from the usual trend, or (ii) when there exists the risk to have quality degradation. The process metrics such as reuse index, RI, and productivity index, PI, can be very useful to estimate the general quality of the development team. In particular, they must be used to maintain under control the efficiency of the team in analyzing and implementing the system under specification. For the same reasons, the trend of the expected system cost, $ESCost$, must also be controlled [Fenton 1991], [Henderson-Sellers et al. 1994].

4.4 Methodologies support

The metric framework allows the estimation of system complexity, size, cost, etc. at each level of abstraction and specification details. This means that most of the metrics proposed can be used during all phases of the software life-cycle according to the new approaches of continuous metrication [Rombach 1990], [Henderson-Sellers et al. 1994]. It should be noted that different values are obtained in different phases of the software life-cycle. The software life-cycle can be regarded as comprised of five phases: analysis, abstract design (i.e., composition/decomposition), basic components design (i.e., implementation of XSM, XCM), testing (i.e., verification and validation), and maintenance. In object-oriented methodologies, the

phases	measurements
Analysis	CC, ECC, $ESCost$, CS, ECS, ECD
Abstract Design	CS, CC, VI, CR, RI, PI
Components Design	CS, CC, VI, CR, RI, PI
Testing	TI
Formal Verification	CR, VI
Validation	ECD, CC
Maintenance	CC, VI, TI

Table 7: Metrics and the phases of software life-cycle.

separation among these phases is not well-defined. In fact, the system under specification can have some parts under analysis while others are under design or testing, etc. The life-cycle can be usually modeled by means of the spiral or the fountain model [Boehm 1986], [Henderson-Sellers et al. 1990], [Booch 1991], [Meyer 1988], [Wirfs-Brock et al. 1990], [Nesi 1994], etc. In both these models, the metric framework proposed can be used as a support for the approach adopted. In fact, both these models allow the process of refinement and are considered as suitable for object-oriented development.

In the course of analysis and design, metrics can be used for helping the designer to generate high quality specifications. This will also guarantee low-costs of testing, extendibility and maintenance of the final product. In addition, further reuse of the whole system or of system components must also be facilitated by specifying understandable and well-verifiable classes. In these phases, the metric framework is able to maintain the specification process under control.

In Tab.7, the metrics which can be used in the phases of the software life-cycle in order to guarantee the product quality and the control of specification development and maintenance have been reported. In this way, the adoption of metrics can be integrated into the methodology; thus, the developer's decisions can be based on objective measurements instead of subjective impressions. Therefore, slightly different (in metrics and thresholds) quality criteria must be set for each phase of the software life-cycle. In addition, special metrics can be defined in order to: (i) measure the production of deliverables (reports and demonstrator) during the whole software life-cycle, (ii) evaluate the costs of reusing class libraries, and (iii) identify the most suitable classes to be reused.

5 Conclusions

A set of metrics for controlling the quality of an object-oriented development process and its integration in the CASE tool named TOOMS [Bucci et al. 1993], has been presented. This integration allows a more effective control of specification quality, and reduces the cost for quality control. The metrics proposed have been defined for a formal object-oriented model and language, mainly focused on the specification of embedded reactive systems (i.e., TROL [Bucci et al. 1994]). These metrics cover all aspects of system specification, helping the TOOMS users to produce well-stated applications. Metrics are a support for maintaining under control the quality of the specification, monitoring class and system complexity, size, testability, reusability, verifiability, etc. This support compels the developers to produce specifications which satisfy the imposed rating and to test the specification when it has grown too much. The specifications are guaranteed to be formally correct by means of the TOOMS facilities for system verification and validation.

```
Class Sample1
    Provided_services: // i.e., inputs
        IA [3.2,4.5]: IA_Type;
        IB : IB_Type;
        IC : IC_Type;
    Required_services: // i.e., outputs
        OA : OA_Type;
        OB : OB_Type;
    Clauses:
        CLA1 : New(IA) → Ready(OB) - - [2.1,4.5];
end;
```

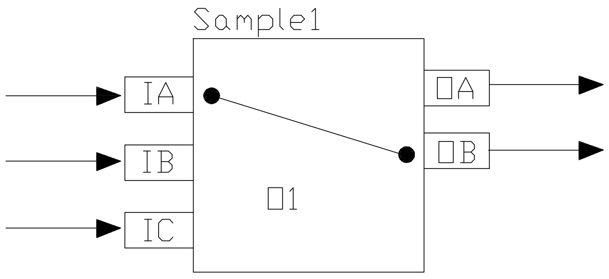


Figure 6: External class description of class Sample1, and visual representation of Object O1 of class Sample1.

Most of the metrics defined in the metric framework are new; while the others are equivalent in general terms to others metrics proposed in the literature. The metrics proposed have been defined to ensure their applicability in all phases of the software life-cycle; in fact, most of them can estimate congruent values even if the system is only partially specified according to the capabilities of many new languages, such as OSDL, TRIO+, and TROL. The TOOMS tool with metrics has been used on several case studies showing the advantages of our model. Most of the concepts proposed (metrics and approaches for their definition), are general enough to be used with other object-oriented approaches with minor changes.

A A Summary of TOOMS/TROL Notation

TROL classes are defined by means of their external and internal class descriptions, which correspond to public and private class members according to the object-oriented paradigm.

A.1 External class description

The external class description (see Fig.6) reports the public features of the class. These are **Provided_services**, **Required_services**, and **Clauses**. Provided and required services can be regarded as input and output ports (gates) (i.e., IO-model [Coleman et al. 1992]).

In the early steps of reactive systems specification, some temporal requirements on system behavior are usually imposed. In the external class description, time is modeled by means of temporal constraints [Dasarathy 1985]. These constraints are associated with services and clauses – e.g., [3.2,4.5] associated with service IA in Fig.6 specifies the minimum and the maximum rate (bounds) of that service in time units. Temporal constraints can also be associated with **Clauses**. Clauses represent a dependence between the services corresponding to the connected IOs (i.e., they are descriptive constraints on class behavior). Referring to Fig.6, clause CLA1 specifies that when a new message arrives at IA, object O1 will make ready a request on OB in the specified time interval. In this case, the constraint associated with clauses describes the bounds for the time of reaction [Dasarathy 1985]. Services are only a static class description in the TROL model, while clauses describe the external class behavior. Since a clause describes the external class behavior, conditions must be expressed only in terms of public references. However, the clauses represent a non-exhaustive description of class behavior; these are very useful during the composition/decomposition process of a class and for the definition of the abstract behavior in the early phases of system specification. The external class description in terms of services and clauses with associated temporal constraints is the instrument by which the class is validated. In addition, an external class description reporting both static

```
Class EstimatorBuffered specializing non_basic_object_class
  Provided_services:
    data1 : DataType;
    flushB : Signal;
    elab : Signal;
  Required_services:
    results : Real;
    err available : EstimatorErrType;
  Clauses:
    ESTIMATION: New(elab) ∧ err==OK → Ready(results);
    FLUSH : New (flushB) → err==EMPTY;
    DATA : New (data1) ∧ err==EMPTY → err==OK;
  /*** private parts ***/
  Attributes:
    B1 : Buffer;
    S1 : Estimator;
  Connections:
    data1 - - B1.datain;
    elab - - S1.eval;
    S1.result - - results;
    S1.err - - err;
    S1.req_data1 - - B1.get;
    B1.dataout - - S1.the_data1;
    B1.is_empty - - S1.buf_st1;
    flushB - - B1.flush;
end;
```

Figure 7: Description of the class EstimatorBuffered with the external description of class Estimator, where EstimatorErrType is defined as an enumeration: **enum** EstimatorErrType {EMPTY,OK};. The description of the class Buffer will be provided later.

and dynamic aspects of the class is a vehicle for (i) understanding the nature of the class without inspecting the whole code, (ii) generating the final patterns for testing the class functionalities [Bucci et al. 1994].

A.2 Internal class description

The internal class description corresponds to the class implementation. In the TROL model, classes can be regarded as *non-basic object classes* or *basic object classes.*

A **non-basic object class** is implemented in terms of a set of communicating sub-objects. This leads to a hierarchical organization of the software structure. When two objects are connected together through their services (required-to-provided, belonging to the same type, e.g., Boolean with Boolean) a channel of communication is established. The default communication model is synchronous on a unidirectional channel, as in [Shaw 1992]. Client/server communications are supported through *message passing*; messages are considered as tokens irrespective of their content – i.e., data, (control) commands or both.

In TROL, there are two kinds of provided services: normal and buffered[2] (**buffered**), and two kinds of required services: normal and always available[3] (**available**). Through the connection of these types of services four types of communication mechanisms are defined: (a) synchronous communication where both sender and receiver are blocked; (b) asynchronous communication with a non-blocked sender and a blocked receiver (when the buffer is empty); (c) asynchronous non-blocked communication corresponding to the request of the latest information value from the sending object (overwriting mechanism); (d) asynchronous non-blocked communication, where a message is stored in the buffer only if it is changed with respect to

[2]implemented by means of a buffer with an infinite dimension.

[3]when the associated information is always there, that is, when the service has its information always ready for the receiver. Available services can be used to present outside the object state.

```
Class Buffer specializing XSM
  Provided_services:
    datain : DataType ;
    get : Signal ;
    flush : Signal ;
  Required_services:
    dataout : DataType ;
    is_empty available : Boolean;
  Clauses:
    GET : New (get ) ∧¬ is_empty → Ready (dataout );
    FLUSH : New (flush) → is_empty;
    DATAIN : New (datain) ∧ is_empty → ¬ is_empty;
  /*** private parts ***/
  Attributes: // auxiliary variables of the state machine
    in : Integer ;
    out : Integer ;
    is_empty : Boolean ;
    Buff : DataType [∞ ];
  States:
    START: { in=0; is_empty=TRUE; out=0; }
    CENTRAL: { }
    WRITEIN: { in=in+1; Buff[in]=datain; is_empty=FALSE; }
    WRITEOUT: { dataout=Buff[out] - - [0,10.3]; out=out+1; }
    ISEMPTY: { is_empty=TRUE; }
  Paths:
    INIT: { START: → CENTRAL;       CENTRAL: New(flush) → START; }
    PUT: { CENTRAL: New(datain) → WRITEIN;       WRITEIN: →CENTRAL; }
    GET: { CENTRAL: New(get) ∧¬ is_empty → WRITEOUT;
        WRITEOUT: in != out → CENTRAL;       WRITEOUT: in==out → ISEMPTY;
        ISEMPTY: → CENTRAL;       WRITEOUT: Fail(TIMEFAIL) → CENTRAL; }
end;
```

Figure 8: Textual specification in TROL of the corresponding visual description in TOOMS for the class Buffer.

the last value.

In Fig.1, the visual representation of class EstimatorBuffered as composed of two sub-objects, S1 and B1, belonging to classes Estimator and Buffer, is reported, as produced by the TOOMS tool BlockEditor. In Fig.7 the descriptive counterpart of Fig.1 is reported.

Basic object classes are those that cannot be regarded as a composite object; and are implemented by using: (i) an extended state machine formalism called “eXtended State Machine” (XSM) model, or (ii) a set of pure data transformations of input into output data (without any notion of state variables) called “eXtended Combinatorial Machines” (XCMs). A TROL environment presents, at the lowest level of hierarchy, a lattice of communicating basic objects which are implemented either as XSMs or XCMs. In Fig.1, the implementation of class Buffer as an XSM is presented as it appears on the TOOMS visual interface. A set of statements (either assignments, and/or procedure calls) is associated with each state, these are executed when the state is reached. In an XSM, temporal constraints can be applied to different predicates, such as the conditions associated with state transitions, requests of provided services, sending of required services, numerical expressions, and procedure calls [Bucci et al. 1994]. In Fig.8, the complete description of the class Buffer is schematized.

It should be noted that, in the XSM model, a basic object class may present a state diagram with a central state – e.g., state named CENTRAL in Fig.1. A number of transitions start from such a state, and each transition identifies a typical branch or aggregate of states called class *path*. For example, the class Buffer in Fig.8 presents three **Paths** (INIT, PUT, GET). In TROL, each class implemented as an XSM is an active object which in turn is implemented with one thread of execution for each path (the concept of path in TROL is equivalent to that of process in SDL). In this way, an object can satisfy concurrent requests of services, if they belong to different paths. These threads (i.e., paths) are signal sensitive, meaning that they are waiting for a signal to change their state; hence, their execution is causal (i.e., event-driven).

The XCMs are a particular case of an XSM in which there exists only a single state, and auxiliary variables do not maintain their values from an execution to the next.

In TROL, the specialized non-basic object classes inherit attributes, provided and required services, connections, and clauses from their super-class according to the monotonic concept of inheritance [Bucci et al. 1994]. Analogous rules are defined for basic object classes that can inherit attributes, provided and required services, states and paths from their superclass.

Acknowledgments

The authors would like to thank Prof. G. Bucci for his valuable suggestions, M. Traversi and A. Borri of CQ_ware for their help, and O.T.E. S.p.A. for providing micro-telephone projects.

References

[Albrecht et al. 1983] A. J. Albrecht and J. E. GaffneyJr. Software function, source lines of code, and development effort prediction: A software science validation. *IEEE Transactions on Software Engineering*, 9(6):639–648, Nov. 1983.

[Basili et al. 1984] V. Basili and D. M. Weiss. A methodology for collecting valid software engineering data. *IEEE Transactions on Software Engineering*, 10(6):728–738, 1984.

[Boehm 1982] B. W. Boehm. *Software Engineering Economics*. Prentice Hall, 1982.

[Boehm 1984] B. W. Boehm. Software requirements economics. *IEEE Transactions on Software Engineering*, 10(1):4–21, Jan. 1984.

[Boehm 1986] B. W. Boehm. A spiral model of software development and enhancement. *ACM SIGSOFT Software Engineering Notes*, 11(4):14–24, 1986.

[Booch 1986] G. Booch. Object-oriented development. *IEEE Transactions on Software Engineering*, 12(2):211–221, Feb. 1986.

[Booch 1991] G. Booch. *Object-Oriented Design with Application*. The Benjamin/Cummings Publishing Company, California, USA, 1991.

[Braek et al. 1993] R. Braek and O. Haugen. *Engineering Real Time Systems: An object-oriented methodology using SDL*. Prentice Hall, New York, London, 1993.

[Bucci et al. 1993] G. Bucci, M. Campanai, P. Nesi, and M. Traversi. An object-oriented case tool for reactive system specification. In *Proc. of 6th International Conference on Software Engineering and Its Applications (sponsored by: EC2, CXP, CIGREF, and SEE)*, Le CNIT, Paris la Defense, France, 15-19 Nov. 1993.

[Bucci et al. 1994] G. Bucci, M. Campanai, P. Nesi, and M. Traversi. An object-oriented dual language for specifying reactive systems. In *Proc. of IEEE International Conference on Requirements Engineering, ICRE'94*, Colorado Spring, Colorado, USA, 18-22 April 1994.

[Bucci et al. 1995] G. Bucci, M. Campanai, and P. Nesi. Tools for specifying real-time systems. *Journal of Real-Time Systems*, page in press, March 1995.

[Card et al. 1990] D. N. Card and R. L. Glass. *Measuring Software Design Quality.* Prenticel Hall, Englewood Cliffs, NJ, USA, 1990.

[Carrington et al. 1990] D. Carrington, D. Duke, R. Duke, P. King, G. Rose, and G. Smith. Object-z: An object-oriented extension to z. In S. T. Voung, editor, *Formal Description Techniques.* Elsevier Science, 1990.

[Coleman et al. 1992] D. Coleman, F. Hayes, and S. Bear. Introducing objectcharts or how to use statecharts in object-oriented design. *IEEE Transactions on Software Engineering*, 18(1):9–18, Jan. 1992.

[Coulunge et al. 1993] B. Coulunge and A. Roan. Object-oriented techniques at work: Facts and statistics. In *Proc. of the International Conference on Technology of Object-Oriented Languages and Systems, TOOLS Europe 93*, pages 89–95, Versailles, France, 8-11 March 1993.

[Dasarathy 1985] B. Dasarathy. Timing constraints of real-time systems: Constructs for expressing them, methods of validating them. *IEEE Transactions on Software Engineering*, 11(1):80–86, Jan. 1985.

[Dürr et al. 1992] E. H. H. Dürr and J. vanKatwijk. Vdm++: A formal specification language for object-oriented designs. In G. Heeg, B. Mugnusson, and B. Meyer, editors, *Proc. of the International Conference on Technology of Object-Oriented Languages and Systems, TOOLS 7*, pages 63–78. Prentice-Hall, 1992.

[ESPRIT-5494 1992] ESPRIT-5494. Ami handbook. Technical report, AMI consortium, January 1992.

[Fenton 1991] N. E. Fenton. *Software Metrics: a Rigorous Approach.* Chapman and Hall, London, 1991.

[Gangopadhyay et al. 1993] D. Gangopadhyay and S. Mitra. Objchart: Tangible specification of reactive object behavior. In O. M. Nierstrasz, editor, *Proc. of 7th European Conference on Object-Oriented Programming, ECOOP'93, Lecture Notes in Computer Sciences Springer Verlag, LNCS 707*, pages 432–457, Kaiserslautern, Germany, July 1993.

[Henderson-Sellers 1991] B. Henderson-Sellers. Some metrics for object-oriented software engineering. In *Proc. of the International Conference on Technology of Object-Oriented Languages and Systems, TOOLS 6 Pacific 1991*, pages 131–139. TOOLS USA, 1991.

[Henderson-Sellers 1992] B. Henderson-Sellers. Modularization and mccabe's cyclomatic complexity. *Communications of the ACM*, 35(12):17–19, Dec. 1992.

[Henderson-Sellers 1993] B. Henderson-Sellers. The economics of reusing library classes. *Journal of Object Oriented Programming*, July-August 1993.

[Henderson-Sellers et al. 1990] B. Henderson-Sellers and J. M. Edwards. The object oriented systems life cycle. *Communications of the ACM*, 33(9):143–159, Sept. 1990.

[Henderson-Sellers et al. 1993] B. Henderson-Sellers, D. Tegarden, and D. Monarchi. Object-oriented metrics. In O. M. Nierstrasz, editor, *Proc. of 7th European Conference on Object-Oriented Programming, ECOOP'93, Tutorial notes*, Kaiserslautern, Germany, July 1993.

[Henderson-Sellers et al. 1994] B. Henderson-Sellers, D. Tegarden, and D. Monarchi. Metrics and project management support for an object-oriented software development. In *Tutorial Notes TM2, TOOLS Europe'94, International Conference on Technology of Object-Oriented Languages and Systems*, Versailles, France, 7-10 March 1994.

[Jensen 1991] R. L. Jensen. Parametric estimation of programming effort: An object-oriented model. *Journal of Systems and Software*, 15:107–114, 1991.

[Jensen et al. 1985]	H. A. Jensen and K. Vairavan. An experimental study of software metrics for real-time software. *IEEE Transactions on Software Engineering*, 11(2):231–234, Feb. 1985.
[Kemerer 1987]	C. F. Kemerer. An empirical validation of software cost estimation models. *Communications of the ACM*, 30(5):416–429, May 1987.
[Laranjeira 1990]	L. A. Laranjeira. Software size estimation of object-oriented systems. *IEEE Transactions on Software Engineering*, 16(5):510–522, 1990.
[Li et al. 1993]	W. Li and S. Henry. Object-oriented metrics that predict maintainability. *J. of Systems Software*, 23:111–122, 1993.
[Mandrioli 1993]	D. Mandrioli. The object-oriented specification of real-time systems. In *Tutorial Note of the International Conference on Technology of Object-Oriented Languages and Systems, TOOLS Europe '93*, Versailles, France, 8-11 March 1993.
[McCabe 1976]	T. J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, 2(4):308–320, 1976.
[Meyer 1988]	B. Meyer. *Object-Oriented Software Construction.* Prentice Hall, C. A. R. Hoare Series, New York, USA, 1988.
[Meyer 1990]	B. Meyer. Tools for the new culture: Lessons learned from the design of the eiffel libraries. *Communications of the ACM*, 33(9):68–88, 1990.
[Morzenti et al. 1992]	A. Morzenti and P. SanPietro. Object oriented logical specification of time critical systems. Technical report, Politecnico di Milano, Dipartimento di Elettronica, Piazza Leonardo da Vinci 32, Milano, Italy, 1992.
[Nesi 1993]	P. Nesi. An object-oriented language and compiler for reactive systems. Technical report, RT 13/93 Dipartimento di Sistemi e Informatica Facolta di Ingegneria, Universita di Firenze, Florence, Italy, 1993.
[Nesi 1994]	P. Nesi. Object-oriented paradigm for the specification of real time systems. In *Objective: Quality 1994*, Milan, Italy, 11-13 May 1994.
[NorthernTelecom 1993]	NorthernTelecom. Objectime: Object-oriented case for real-time systems. Technical report, Bell-Northern Telecom, 1993.
[Panaroni et al. 1990]	P. Panaroni and A. Musone. Design software metrics for the hood method. Technical report, INTECS SISTEMI spa, Pisa, Italy, 1990.
[Rask et al. 1993]	R. Rask, P. Laamanen, and K. Lyytinen. Simulation and comparison of albrecht's function point and demarco's function bang metrics in a case environment. *IEEE Transactions on Software Engineering*, 19(7):661–671, July 1993.
[Rombach 1990]	H. D. Rombach. Design measurement: Some lessons learned. *IEEE Software*, pages 17–25, March 1990.
[Sanden 1989a]	B. Sanden. The case for eclectic design of real-time software. *IEEE Transaction on Software Engineering*, 35(3):360–363, March 1989.
[Sanden 1989b]	B. Sanden. An entity-life modeling approach to the design of concurrent software. *Communications of the ACM*, 32(3):330–343, March 1989.
[Shaw 1992]	A. C. Shaw. Communicating real-time state machines. *IEEE Transactions on Software Engineering*, 18(9):805–816, Sept. 1992.

[Stetter 1984]	F. Stetter. A measure of program complexity. *Computer Language*, 9(3):203–210, 1984.
[Thomas et al. 1989]	D. Thomas and I. Jacobson. Managing object-oriented software engineering. In *Tutorial Note, TOOLS'89, International Conference on Technology of Object-Oriented Languages and Systems*, page 52, Paris, France, 13-15 Nov. 1989.
[Warburton 1983]	R. D. H. Warburton. Managing and predicting the costs of real-time software. *IEEE Transactions on Software Engineering*, 9(5):562–569, Sept. 1983.
[Wearing 1992]	A. Wearing. Software development metrics for real-time embedded systems. Technical report, Evisa System, Disley, Cheschire, SK 12 2BH, UK, 1992.
[Wirfs-Brock et al. 1990]	R. J. Wirfs-Brock, B. Wilkerson, and L. Winer. *Designing Object Oriented Software*. Prentice Hall, Englewood Cliffs, N.J., USA, 1990.
[Yap et al. 1993]	L.-M. Yap and B. Henderson-Sellers. Consistency considerations of object-oriented class libraries. Technical report, University of New South Wales, Information Technology Research Centre, n.93/3, Sydney, Australia, June 1993.
[Zuse 1994]	H. Zuse. Quality measurement – validation of software metrics. In *Proc. of the 7th International Software Quality Week in San Francisco, QW'94*, pages 4–T–2. Software Research, 17-20 May 1994.

Contents

Biographies

Paolo Nesi received the degree in Electronic Engineering from the University of Florence, Italy. In 1992, he received the Ph.D. degree in Electronic and Informatics Engineering from the University of Padoa, Italy. In 1991 he was a visitor at the IBM Almaden Research Center, CA, USA. Since November 1991 he is with the Department of Systems and Informatics of the University of Florence, Italy, as a Researcher and Assistant Professor of both "Computer Science" and "Software Engineering". Since 1987 he is active on different research topics, real-time systems, formal specification languages, software metrics, parallel architectures, physical models, image processing. He has published more than 50 papers on international journals and conference proceedings. Dr. P. Nesi is an editorial board member of the *Journal of Real-Time Imaging*, Academic Press, and he is also project manager of OFCOMP (MEPI DIM45 ESPRIT III) for the University of Florence. Dr. P. Nesi is a member of IEEE, IAPR, and AIIA.

Maurizio Campanai received the degree in Electronic Engineering in 1991 from the University of Florence, Italy. In 1992 he was with the Department of Systems and Informatics of the University of Florence, Italy. Since 1993, he is the technical manager for the CESVIT (Center for the Technical Improvement of Industries) of CQ_ware (Center for Software Quality). His main interests and research areas include software engineering, visual languages, and software quality assurance. Dr. M. Campanai is a member of IEEE.