

Measures and Procedures: Lessons Learned from the CLARAty Development at NASA/JPL

Hari D. Nayar and Issa A.D. Nesnas

Jet Propulsion Laboratory
4800 Oak Grove Drive, M/S 82-105
Pasadena CA 91001, USA

Hari.D.Nayar@jpl.nasa.gov, Issa.A.Nesnas@jpl.nasa.gov

Abstract— We propose three categories of measures and procedures for this Workshop on Measures and Procedures for the Evaluation of Robot Architectures and Middleware. The categories are *Programmatic*, *System-Engineering* and *Component-Specific*. Within these categories, we suggest measures and procedures that we have identified from our experience with the CLARAty reusable robotic software.

I. INTRODUCTION

OUR selection of measures and procedures for the evaluation of robotic software systems has been strongly influenced by our experience with the CLARAty system (Volpe 2001, Nesnas 2003, CLARAty 2005, Nesnas 2006). CLARAty is an on-going development at JPL and collaborating NASA centers and universities with a legacy of over ten years of development. It is a framework for reusable robotic software. At its lowest level, CLARAty implements software abstractions for hardware interfaces in an object-oriented hierarchy. Upon this hardware abstraction layer, re-usable software components are built to interface to higher levels of control. As a result, software to implement complex behavior and sophisticated operations is platform independent. Examples of such capabilities include pose estimation, navigation, locomotion and planning. In addition to supporting multiple algorithms for each capability, CLARAty provides adaptations to multiple robotic and rover platforms. CLARAty is a domain-specific robotic architecture designed with four main objectives:

1. To promote the reuse of robotic software infrastructure across multiple research efforts
2. To promote the integration of new technologies developed by the robotics community onto rover platforms
3. To mature robotic capabilities through reuse and enable independent formal validation
4. To share the development with the robotic community to promote rapid advancement and leveraging of capabilities

Development of the infrastructure to support these objectives is continuing in many directions including improved interfaces to actuators and sensors, camera modeling and image processing, mechanism modeling,

locomotion, pose estimation, navigation and interfaces to higher level planners.

The areas of robotics covered within the CLARAty system include hardware interfaces, kinematics, manipulation, mobility, estimation, control, vision, navigation & path planning, and artificial intelligence planning.

II. CATEGORIES OF MEASURES AND PROCEDURES

We have taken a very general interpretation of the word *measure* in the context of this discussion. *Measure*, in this paper, includes any method of characterizing a robot software or middleware system. In many of our cases, measures are descriptive rather than quantitative. For these cases, where possible, we have attempted to be more precise and objective by listing *measurements* as a selection from a list of choices. Please see item 1 in Table I for an example.

It is useful to categorize measures and procedures for robot software architectures in order to characterize them. While there are alternative, possibly better, categories, we have selected the following three: *Programmatic*, *System-Engineering* and *Component-Specific*. Our categorization is organization-centric. *Programmatic* measures are mainly of interest to management, *system-engineering* measures are of interest to the system engineer and the *component-specific* measures are of interest to the developers of the component. This classification is adequate for our discussion because it helps organize our measures and assigns responsibility for them to a person in the development team hierarchy. There are other possible categorizations. Fenton (2000) lists *Products*, *Processes* and *Resources* as an approach used for categorizing software measures. *Structural*, *Code Metrics* and *Hybrid* are the categories used by Kafura(1985). A potential outcome from this Workshop, in addition to identifying measures and procedures, could be a categorization that the community agrees on.

TABLE 1A
PROGRAMMATIC MEASURES AND PROCEDURES: ARCHITECTURE

#	Meas./Proc.	Description	Measure Criteria
1	Architectural Approach	<p>A number of architectural approaches are possible including:</p> <ol style="list-style-type: none"> 1. Abstract Model Approach: uses abstractions as first order elements in the architecture. Uses a hierarchy of abstract and concrete models to represent logical and physical devices and capabilities. Uses technologies from object-oriented design, component design, and generic programming. (add CLARAty example, Control Shell, OROCOS) 2. Data Centric Approach: uses data as first order elements in the design. Uses public and subscribe mechanisms in a data distributed system (add reference) (add DDS example). 3. Service-Oriented Approach: uses services as first order elements. Services are language independent, computationally distributed, and stateless (state and intent) services as first order elements in the architecture. For example, in MS Robotics Studio (reference) state and intent come into a component through an XML document which process and generates an XML output document. 4. Other Approach: approach does not fall within above categories. <p>This item may, alternatively, be grouped in the <i>System engineering</i> category.</p>	Multiple choice from one of the options
2	Programming Paradigms and Languages	<p>Two primary programming paradigms have been used in robotics: (these can also go into the Architectural approach)</p> <ol style="list-style-type: none"> 1. Procedural programming paradigm: This paradigm uses program logic to define the sequence of robot actions. This paradigm is most prevalent within the robotics community. 2. Declarative programming paradigm: This paradigm defines the robot actions using pre- and post-conditions, and then uses a search engine to generate the program logic. This paradigm is most prevalent within the Artificial Intelligence robotic community (e.g. CASPER 2007). <p>There is also the choice of the programming language based on the selected paradigm. Within procedural languages there are functional programming languages like C or Fortran and object-oriented programming languages like C++ or Java. With the declarative programming paradigm, most languages are custom ones developed at research labs.</p>	Multiple choice. Language choice
3	Architectural Heterogeneity	<p>A measure of the heterogeneity of the architectural paradigms and programming languages that are used:</p> <ol style="list-style-type: none"> 1. Low Heterogeneity: one dominant architectural paradigm and a single programming language. 2. High Heterogeneity: multiple architectural paradigms and multiple programming languages. 	Multiple choice
4	Deployment Architecture	<p>There are a number of parameters that characterize the deployment architecture of a robot software system. These include:</p> <ol style="list-style-type: none"> 1. Computational architecture <ol style="list-style-type: none"> a. Single centralized node b. Distributed homogeneous or heterogeneous nodes 2. Operating system type: <ol style="list-style-type: none"> a. Soft real-time b. Hard real-time 3. Type and scale of processors: <ol style="list-style-type: none"> a. Integer-based processor vs. processors with floating point support b. X86 family, PowerPC family, SPARC family, etc. 4. Type of compilers to be supported 	Multiple choice from one of the possible combinations of these options.

TABLE 1B
PROGRAMMATIC MEASURES AND PROCEDURES: SOFTWARE DEVELOPMENT

#	Meas./Proc.	Description	Measure Criteria
5	Development Environment	There are a number of integrated development environments (IDE) available for software development. Examples are Eclipse and Visual Studio. Developers may, alternatively, use other home-grown environments by combining components needed for development. IDEs can have an affect quality and productivity of the software development and testing process.	Multiple choice.
6	Code Organization	Code organization describes software system decomposition to modular units. There are two levels of code organization. The first is the organizational structure within the execution environment to facilitate modularity, encapsulation and code re-use. This is discussed in greater detail under <i>Functional decomposition</i> under <i>System-Engineering</i> in Table II. The second level is the organization within the development environment. Options include decomposition by function, by developer and other source. The organization may be in a flat or hierarchical structure. There are probably as many approaches for organization of robot software as there are implementations. Choices made in some of the items listed above like architectural approach, deployment architecture and one or more languages used will influence the organization of the software.	Descriptive measure
7	Coding Standards	Coding standards define how the software will be written. This is useful because it helps unify the format of the written software and facilitates sharing software among developers. Standards (for example on use of exception handling or function return types) also help maintain a uniform level of quality throughout the code. The ANSI ISO/IEC14882 standard [ANSI 1988] is an example standard that may form the basis of a team's coding standard.	Reference to ANSI or other standard.
8	Documentation	Documentation of software is an extremely important element of the software production process. It is a means of communication within the development team, for users and an information repository to capture the development effort for future use and maintenance (Sommerville, 2002). There are many categories of documentation ranging from high-level user documentation, technical publications of algorithms, to low-level code comment documentation. Consequently, there can be many measures to quantify documentation. These include percentage of lines of documentation in source code, effectiveness of the documentation, maintenance and correctness of documentation especially through software revisions, number of journal or conference publications and so on. Automated documentation procedures can be incorporated into the software development with little effort, for example, with tools like Doxygen.	Multiple measures (see description).
9	Developer-Coordination Procedure	Procedures for coordination of multiple and possibly disparate developer teams are critical for successful integration, testing and deployment of robotics software. Developer coordination procedures include: developer training, meetings and tele-conferences, coordinating exchange visits with software deliveries, maintaining mailing lists, announcing software commits and releases, and maintaining a website for documenting development procedures, status, and system information.	

TABLE 1C
PROGRAMMATIC MEASURES AND PROCEDURES: SOFTWARE QUALITY

#	Meas./Proc.	Description	Measure Criteria
10	Coherence	Although hard to quantify, we suggest this measure to indicate the importance of developing software that adheres to, and efficiently embodies its design philosophy. An example, taken from the CLARAty development, is the attempted design of the class structure at the abstract motor level to be also reflected at the abstract locomotor level. This measure attempts to capture the consistencies (or inconsistencies) in design patterns between architectural elements throughout system. A suggested quantification of this measure is to enumerate the instances where the design philosophy is not followed.	Number of violations of design philosophy.
11	Code Size	The number of source lines of code (SLOC) has been proposed as a measure of the size of a software package. Physical SLOC counts the total number of lines in the software while logical SLOC is the number of statements in the package.	Physical and Logical SLOC.
12	Complexity	There are many possible metrics that can capture the complexity in a software package. All the following increase the complexity in software: <ol style="list-style-type: none"> 1. Number of processor nodes 2. Combinations of different processor types (for example, PPC and x86 vs. only PPC or only x86, or x86 with embedded micro processors that you write firmware for), 3. Use of more than one programming language (linear or non linear) 4. Variety and content of information in an algorithm 5. Number of algorithms in the system, 6. Number of sources of algorithms in a system (i.e. number of developers that are collocated and number of distributed developers) 7. Choice of complex vs. simple algorithmic solutions (for example, closed form vs. numeric solutions for kinematics or dynamics) 8. Number of sensing modalities 9. Amount of effort that has gone into the development (measured as the number of work hours). 10. The Cyclomatic number (McCabe 1976). 	Multiple measures (see description).
13	Software Validation and Verification Procedures	To maintain the quality of new components integrated in to a software system, procedures for design review, implementation process, validation, verification, and maintenance are needed. These procedures help ensure that the component complies with the system design standards, meet the desired interface requirements, are implemented correctly and adhere to the development policies.	Matrix checklist for to ensure all proper procedures are followed
14	Regression Testing Procedure	Measures to evaluate regression testing include indicating if an automated process exists, enumeration of unit test coverage, whether it is multi-target, test frequency, consistency of the implementation of unit tests, consistent report of results, and memory leak checking.	Multiple measures (see description).

We list metrics and procedures from our experience with the development of the CLARAty robot software architecture. Our list is not exhaustive – there will be items that overlap and are missing in comparison to items from other groups participating in this Workshop.

The reader will notice that there is overlap among some items listed. For example, a measure of software system complexity has some overlap with the deployment

architecture (centralized single processing computing versus distributed multiple-processor computing). We will not attempt to identify or quantify these overlaps in this paper.

A. Programmatic

Programmatic measures and procedures are items of interest to the manager, systems engineer, or end user of the robotic software architecture. They provide: 1) a common language for describing the overall robot software system for comparison against other software systems, 2) quantities to

TABLE IIA
SYSTEM-ENGINEERING MEASURES AND PROCEDURES

#	Meas./Proc.	Description	Measure Criteria
1	Functional Decomposition	The functional decomposition of robotics software can be done in many ways. The architectural design will have a strong influence on the decomposition of the software. Two decompositions at opposite ends of the spectrum that reflect different architectural styles are: <ol style="list-style-type: none"> 1. A flat-structure with groupings of signals, processing blocks, control models, and finite-state machine models, 2. An object-oriented hierarchical models with utilities and hardware abstraction objects at the lowest level and building up to high-level user interfaces at the top-level. 	Descriptive measure.
2	Access Levels	This measure will answer the question: “Does the architecture allow for access at different levels of granularity, and if so, how many and at what levels?” Possible access levels include at the digital I/O, motor, motor group, locomotor, navigator or robot levels. This is useful because it tells us the levels at which one can interface to hardware and the level at which reuse can occur without the overhead of unnecessary software. Does the architecture provide an API for a motor, camera, camera group, IMU, digital I/O, etc, navigator, locomotor, etc.	Number of access levels, descriptive measure of levels.
3	Sub-system Coverage	Robotics includes technology from many different disciplines. Furthermore, the technology itself is expanding with the rapid development of new innovations. No robotics software system can include all possible technologies. However, common sub-systems that are used in many robotics software systems can be identified. These include (and may be further subdivided): vision, locomotion, manipulation (serial, parallel, hybrid), pose estimation, navigation, trajectory generation, motor control, I/O, and math utilities. A measure we suggest is to draw up a categorization of these sub-systems (it may have a hierarchical structure) and indicate the coverage of a software system over the structure.	Sub-system coverage percentage.
4	Separation of physical and the logical hierarchy	The effort needed to integrate a new low-level hardware device into a robot software system without disrupting its high-level software is a useful measure of how well-designed and adaptable the software is. Another measure that could capture a similar capability is the number of hardware devices performing similar functions but with different interfaces that have been implemented in the software system. For example, for motor control, there are a variety of hardware-dependent motor control architectures based on centralized, distributed, or other configurations. In addition, motor control may be performed on a CPU, with specialized motor control chips (LM629, HCTL1100, etc.) or COTS boards. A clean separation between classes and drivers for a particular hardware device and a generic API layer facilitates easy incorporation of new devices without changing the existing software interfaces.	Effort to implement new hardware device. Number of different device interfaces implemented.

measure the quality and efficiency of the overall software development approach, and 3) procedures to manage and improve the quality of the software development process. Many of these items are relevant for any large software system. Some are particularly relevant to robot software systems. We have grouped these measures and procedures into three categories related to: (1) system architecture, (2) software development, and (3) software quality. Table IA – IC describe items in each of these categories.

B. System Engineering

In the *System-Engineering* category are measures and procedures of interest to the systems engineer, robot software developer, or the expert (or power) user. These items are metrics and procedures related to technical capabilities, to sub-system design, interfaces between sub-systems or approaches implemented. Items in this category are

TABLE IIB
SYSTEM-ENGINEERING MEASURES AND PROCEDURES

#	Meas./Proc.	Description	Measure Criteria
5	Interface Stabilization	Interfaces between components within the robot software system can be designed to minimize changes needed on the other side of the interface when software on one side is changed. This is done in CLARATy with the use of complex data types. For example, the <i>Camera</i> class uses the <i>Camera_Image</i> data type for its argument in the <i>acquire</i> function rather than using raw data types such as (<i>int * data, int nrows, int ncols</i>). Using raw data types makes implicit assumptions about the type of image and its pixel content. It will be useful to have a measure to capture how stable the interfaces are in the software. One possibility for this measure is the number of API changes for every revision. Another is the conciseness (number of arguments used) and ease of use of the APIs.	Number of API changes per revision. Conciseness and ease of use of the APIs.
6	State Management	The following questions can help assess state management in the software: <ol style="list-style-type: none"> 1. Is state dealt with in a consistent manner throughout the system? 2. Is state logging dealt with in a consistent way? 3. Does the system have mechanisms to synchronize state updates? 4. Does the system provide mechanisms to update different states at different rates? 	Yes/No answers to listed questions.
7	Uncertainty Representation	Stochastic representation of information is useful in robotic systems because there is often much uncertainty in the models of the environments that robots operate in. The following two questions give basic measures of how well a system addresses this capability: <ol style="list-style-type: none"> 1. Does the system have a means to represent uncertainty? 2. Does it support and interoperate more than one type? 	Yes/No answers to listed questions.
8	Shared resource handling	Some resources in robotics systems are shared among two or more processes. Examples include memory, hardware devices, power, hardware busses, and computational time. We suggest these questions as a method of measuring how well a software handles shared resources: <ol style="list-style-type: none"> 1. Does the system support multiple clients accessing a shared resource? 2. Does the system support reasoning about shared resources (e.g. queries about current resource state)? 3. Does it support queries on planned usage (how much resource usage do motors in an arm use for a given trajectory)? 	Yes/No answers to listed questions.

especially important because robotic software development is a highly multi-disciplinary field and most applications require the integration of sub-systems from multiple disciplines and developer teams. Detail descriptions of items in this category are listed on Table IIA – IIB.

C. Component-Specific

Component-Specific measures and procedures related to particular implementations of capability, within a specialty or field of robotics, are included in this category. These measures and procedures are applied to capabilities implemented within a sub-system and are of interest to the sub-system developers and system-engineers. There are some general measures and procedures that apply for software in this category. An example of a measure that is general is the computational time for an algorithm. This measure may be applied to an inverse kinematics or a stereo-vision algorithm.

However, there are many more measures and procedures that are unique for a particular field of robotics. An example is the error between the results of a pose estimation algorithm and ground truth. This measure is only relevant for a pose estimation algorithm. Furthermore, it is only measurable under a particular set of laboratory conditions. Some measures in this category are listed on Table IIIA – IIIB.

III. CONCLUSION

Robot software systems are generally very complex. There are some areas within robotics that are easy to standardize and be able to quantify. However, many areas of robotics are difficult to generalize and precisely quantify. Despite the challenge, developing quantifiable measures and procedures for robot software systems will lead to a number of benefits for the robotics community. These include improved

TABLE IIIA
COMPONENT-SPECIFIC MEASURES AND PROCEDURES

#	Meas./Proc.	Description	Measure Criteria
1	Generality	There is a trade-off in modeling effort and computational cost between algorithmic generality and customization for particular applications. A simple example is the inverse kinematics of a serial-link robot arm. We can write a general-purpose algorithm to numerically solve for the inverse kinematics of all serial-link robot arms or write an algebraic solver that solves for particular configurations of robot arms. In robotics, generality is often relative – an algorithm that handles serial and parallel robot arms is more general than one for only serial arms. And an algorithm for hybrid serial and parallel arms would have a higher generality metric. Handling multiple end effectors simultaneously would add even more generality, and so on. For specific areas of robotics, we can state the level of generality with respect to the types of systems it can handle and the restrictions (assumptions) for particular algorithms. It should be noted that more general is not necessarily better because it will be at the cost of computational cost and development effort.	Level of generality – specific for each area of robotics.
2	Numerical Precision/Accuracy	When comparing alternative approaches that perform a similar function, for example a dynamics simulation, it's possible to numerically measure output and compare differences. In some cases, say in pose estimation or stereovision, it's possible to compare estimation results against ground truth. For measurement purposes, a reference example may be developed to compare alternative approaches.	Field-specific numerical result for a reference example.
3	Computational Cost/Efficiency	Computational cost is a measure that can be quantified for particular implementations of algorithms on particular platforms.	Computational cost
4	Actively Maintained	Actively maintained components are modules in the software that continue to be updated. Updates improve the software, adapt it for more platforms, increase computational speed, and so on. There is a cost, however, in maintenance (see below) of active components.	Is a component frozen or active.

development processes and resulting quality, methods for predicting schedules and identifying risks, and metrics for comparing alternative approaches.

ACKNOWLEDGEMENT

This work was carried out at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration.

REFERENCES

- [1] ANSI International standard ISO/IEC 14882, International Standard for C++, 1998.
- [2] CASPER (2007), <http://casper.jpl.nasa.gov>
- [3] CLARATy (2007), <http://claraty.jpl.nasa.gov>
- [4] Fenton, N. E. & Neil, M., (2000) Software metrics: roadmap, Proceedings of the Conference on The Future of Software Engineering, p.357-370, June 04-11, 2000, Limerick, Ireland.
- [5] Kafura, D., (1985) A survey of software metrics, Proceedings of the 1985 ACM annual conference on The range of computing : mid-80's perspective: mid-80's perspective, p.502-506, October 1985, Denver, Colorado, United States.
- [6] McCabe (1976) Complexity Measure, IEEE Transactions on Software Engineering, Volume 2, No 4, pp 308-320, December 1976.
- [7] Nesnas, I.A., (2006) chapter in the *Software Engineering for Experimental Robotics*, Springer Tracts on Advanced Robotics, edited by Davide Brugali, 2006.
- [8] Nesnas, I.A., Wright, A., Bajracharya, M., Simmons, R. & Estlin, T. (2003). CLARATy and Challenges of Developing Interoperable Robotic Software, International Conference on Intelligent Robots and Systems (IROS), Nevada, October 2003.
- [9] Sommerville, I., Software Documentation, in Software Engineering, Vol 2: The Supporting Processes. R. H. Thayer and M. I. Christensen (eds), Wiley-IEEE Press, [Book chapter], 2002.
- [10] Volpe, R., Nesnas, I.A.D., Estlin, T., Mutz, D., Petras, R. & Das, H. (2001). The CLARATy Architecture for Robotic Autonomy, Proceedings of the 2001 IEEE Aerospace Conference, Big Sky Montana, March 2001.

TABLE IIIB
COMPONENT-SPECIFIC MEASURES AND PROCEDURES

#	Meas./Proc.	Description	Measure Criteria
5	Level-of-integration	<p>The CLARAty software has developed four levels to quantify the level of integration of a software module. These are:</p> <ul style="list-style-type: none"> I. Has been deposited into the CLARAty software repository as a stand-alone package, with test software and user documentation. II. Interacts with other components in CLARAty, runs on a robot platform but does not use CLARAty APIs. III. Runs on all CLARAty robot platforms, has no 3rd party undocumented dependencies and meets CLARAty API standard IV. Reviewed by development team, meets CLARAty conventions, uses all relevant CLARAty classes, provides access to internal data and is maintained with other CLARAty software. <p>Note that internally developed modules often start at Level II to III while modules delivered from collaborators will often start at Level I.</p>	Level as defined.
6	Validation and Verification	<p>Defining a validation procedure for an algorithm helps ensure that it meets the requirements of an application. Some algorithms are easier to validate than others. For example, we can determine if a matrix inverse algorithm is correct but it is harder to automatically validate a locomotor move command.</p>	Component specific measure.
7	Management Overhead	<p>There are a number of factors that affect how easy it is to maintain a software component. Maintenance is easier if it is well integrated, and has automated validation and verification procedures.</p>	Combination of previous three measures.
8	Component-specific measures and procedures	<p>There are many field-specific measures and procedures that may be relevant for particular fields but not for others.</p>	Field and algorithm specific.