INFORMATION CONSISTENCY CHECKING IN DOCUMENTATION DRIVEN DEVELOPMENT FOR COMPLEX EMBEDDED SYSTEMS

Valdis Berzins
Department of Computer Science
Naval Postgraduate School
Monterey, CA 93940, USA
berzins@nps.navy.mil

Ying Qiao
Department of Computer Science
Naval Postgraduate School
Monterey, CA 93940, USA
yqiao@nps.navy.mil

Luqi
Department of Computer Science
Naval Postgraduate School
Monterey, CA 93940, USA
luqi@nps.navy.mil

Abstract – **Complex embedded systems, especially systems of embedded systems (SoES) need documentation to support their development. In our research, we are developing a documentation driven development method for SoES. In this method, keeping high confidence properties consistently identified in documentation of different development phases is an important issue since it is critical to ensure software quality of the end product. To address this issue, in this paper we investigate a method for information consistency checking in documentation driven development for SoES. We present an attributed object graph model to describe the semantics of document elements. Based on this model, we show how a set of attribute computation rules can analyze consistency between the key information such as timing properties transformed from one development phase to another.**

1. INTRODUCTION

A. Background

Complex embedded systems that are widely used today are usually deployed for long periods of time. They usually have mission critical requirements and demand real-time and high-confidence performance. These complex embedded systems, known as systems of embedded systems (SoES)[1], are composed of component systems that were developed by different organizations with different tools and run on different platforms. Furthermore, they must rapidly accommodate frequent changes in requirements, mission, environment, and technology. These traits make software development for systems of embedded systems face several challenges. First, key properties of embedded systems, such as high-confidence properties are hard to keep consistent during the whole development process, making software quality difficult to ensure in the end product. Second, a wide variety of stakeholders (sponsors, developers, users, maintainers, etc) are involved in the overall lifecycle of the software. Inconsistent information among different stakeholders is one of the main factors resulting in design faults. Third, complex embedded systems are difficult to evolve and maintain because of the independent development of their constituents and frequent changes in circumstances.

Previous research on embedded system development revealed that documentation plays a crucial role in coping with the above challenges throughout the software life cycle. According to the FIPS PUB 105 definition, documentation refers to all information that describes the development, operation, use, and maintenance of computer software. This information is in a form that can be reproduced, distributed, updated, and referred to when it is needed [2]. Furthermore, software documentation should provide information to support all software life cycle processes, most notably, requirements gathering, quality assurance, design, system evolution and reengineering, project management, communication among all system stakeholders and communication with software tools.

B. Related Work

Software Engineering aims to improve software quality and productivity by providing systematic, disciplined and quantifiable approaches to software development. Documentation has been proven to play a key role in software engineering. Many theories, methods, and techniques related to documentation have been developed in the past decades. There are different specific documents associated with different development phases. Typical phases in the software life cycle include requirements analysis and definition, architectural design, implementation, composition, deployment, maintenance and evolution.

In the requirement phase, a requirement definition, which is a kind of documentation, serves as a starting point for the whole software development process. Natural language is the most common form of requirement definition [3]. By modeling and formalizing the requirement definition, the formal documentation – the requirement specification – can be derived. In this case, the requirement specification is usually written in formal language. Typical examples include [4], [5], [6] and [7]. They use temporal logic to represent the formal requirement specifications that further serve as the basis for verification and validation.

The most important documentation used in the design phase is design specification. This acts as a blueprint for the actual coding by outlining the logic of individual code modules. It also assists maintenance programmers as they modify the program to add enhancements or fix errors [8]. A design specification is generally described by formal or semi-formal methods, such as hierarchy charts, logic charts, state transition diagrams, state machines, data flow diagrams, data dictionaries, object-oriented approaches, and a great number of formal languages [9]. Some typical formal and semi-formal notations used for design specification include UML [10, 11] and some kinds of architecture description language [12, 13]. Prototype system description language (PSDL) [14, 15] is another typical design specification language for real-time embedded systems. It uses operators and data streams between operators to model the embedded systems and captures timing constraints and control constraints of embedded systems. PSDL also provides a graphic interface to stakeholders. In addition, design specification also serves as the basis for formal analysis as described in [16], [17] and [18] to find design faults early in development.

Configuration is another important aspect of software development that is done based on documentation support, such as architectural specification and component specification. In complex control systems, the configuration of components must be flexible enough to allow rapid online reconfiguration and adaptation to react to environmental changes and unpredictable events at run-time. For this purpose, an open software architecture [19] has been used for integrating control technologies and resources.

Although a lot of effort has been applied toward improving documentation technology [8, 20, 21], there are still open challenges that hinder documentation from providing efficient support for complex systems of embedded systems development. First, according to the traditional concept, software documentation consists only of informal text and diagrams intended for human consumption. This kind of static information simply records some results and process steps during the software development. It cannot capture the dynamic information during the development process. Second, keeping documentation up-to-date is difficult and time consuming. The various representations of documentation increase the complexity of maintaining information consistency, increase the intellectual burden on stakeholders, and introduce the need for transformations that are tedious and error prone when carried out manually. Some formal representations with rigorous logic are conducive to machine manipulation but are difficult for human understanding. Informal representations such as natural language are comfortable for many system stakeholders but are too vague and ambiguous for direct use by computer tools. Although multiple views of the information can alleviate this problem, how to maintain consistency among information presented to both the humans and computer tools is still a challenge. In addition, to guarantee software quality in the end product, the information should be kept consistent among documents of successive development phases. Traditional documentation technologies do not solve this problem.

To attack above problems and enable documentation to provide more effective support for complex SoES development, we proposed a documentation driven development method for SoES [22]. This is a new approach for documentation that can enhance integration of computer aided software development methods, encompass the entire life cycle, support system evolution and improve communication with system stakeholders. In this method, keeping consistency of information transformed between successive development phases is an important issue. It is critical for ensuring high confidence in the end product. For this purpose, a specific method is needed to enable the key information to be consistently transferred between documentation of successive development phases. This paper presents such a specific method.

Much research has been done on attribute grammars that constitute a classic technology for compiling [23-26]. An attribute grammar is a specification of computations and dependence based on a formal calculus introduced by Knuth [27]. Since it is an efficient way to handle the semantics of context-free languages, we plan to extend and exploit it to deal with the information consistency issue identified above. In this paper, we present an attributed object graph model to represent aspects of the "meaning" of document elements and use a set of attribute computation rules to analyze and ensure the consistency of information transformed between successive development phases.

C. Organization of This Paper

The rest of paper is organized as follows: Section 2 addresses the core of the documentation driven development method – repository representation; Section 3 presents an attributed object graph model for document elements; Section 4 illustrates the use of attribute computation rules to help ensure consistency of documentation and section 5 presents the conclusion and future work.

II. REPOSOTERY REPRESENTATION

The repository representation is the core of the documentation driven development method. All the information related to development process is stored as knowledge in the documentation repository. Each development phase has its own area in the documentation repository. The information is transformed between different documentation areas that belong to successive development phases. Typical examples of the information stored in the repository are requirement specifications, abstracted models, stakeholder input (from sponsors, end users, developers, technical supporters, etc.), design rationale, project management information and the source code. The repository uses a structured central representation for this knowledge so that different stakeholders can communicate with each other based on consistent information and this knowledge can be consistently transformed between successive development phases. Fig. 1 illustrates the repository representation.

Fig. 1 shows that the repository representation includes three kinds of artifacts, i.e., document elements (DEL), a set of syntactic templates and a set of attribute computation rules. A document element is a basic building block consistent with the semantics of the information contained in the documentation. It is described by a semantic document model. This model is an object model for the information contained in the documentation whose instances form an attributed object graph. The documentation elements are the nodes of this graph. The amount of information associated with each node depends on the degree of formalization for each documentation type. Formal representations have explicit structure at a fine granularity and very simple information associated with individual documentation elements. Informal representations have only a large granularity structure and can have lengthy annotations attached to the nodes. Document elements hold the key information extracted from all the requirements, models, activities and processes related with system development. The model is strongly typed and structured according to a documentation schema. Further development of this approach will need better computer-assisted methods for resolving the ambiguities common in informal representations, transforming them into more formal, finer-grained representations, and for checking the validity of this process.

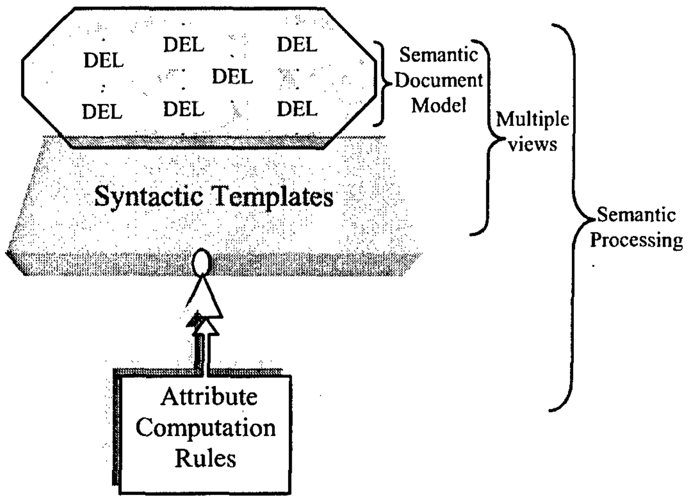

Fig. 1 Repository Representation

Syntactic templates are object operations with parameters. The purpose of a syntactic template is to materialize the part of a specific documentation view that corresponds to a given documentation element. The parameters represent the relevant properties of the context and the descendent nodes of the documentation element. Syntactic templates are designed together with specific sets of rules that govern the manipulation of the data stored in the document elements. The content of the document elements is treated as repository knowledge and the different templates govern how that knowledge is used and presented to the stakeholders and tools in the computer development environment. The combination of a document element and different syntactic templates forms the multiple view presentation of the same information. Combining document elements with corresponding templates can also transform the information between representations written in different description languages [22].

Attribute computation rules represent the methods for computing derived document attributes. They make the repository into an active project support system. These rules are organized in a rule base. The rule base is designed to be open in the sense that new rules can be added without changing the effect of any complete subset of the previous version of the rules. This property supports reliable incremental extension of the automation support provided by the repository and enables steady improvement of decision support processes.

In the long term, the repository will perform a variety of automated and computer aided functions such as the following:

- Materialize external representations of documents suitable for particular stakeholders or tools
- Find appropriate subsets and projections of the documents suitable for particular purposes
- Extract computed attributes of documents, such as expected completion date of the project
- Transform data among different representations as needed to support integration of development processes and tools
- Configuration management of the documents [28-30]
- Project management based on management documents such as plans and schedules [31-33]

To address the problem of consistent information transformation between documentation of successive development phases, we describe the attributed object graph model and attribute computation rules in the following sections.

III. ATTRIBUTED OBJECT GRAPH MODEL

This section explains the computational semantics of the attributed object graph model. This is an object model of knowledge in the documentation repository. It has a nested structure with potentially shared nodes, i.e., directed acyclic graph structure. This representation is a generalization of abstract syntax trees and is designed to represent and efficiently analyze constructs that appear in more than one context. This is a common pattern in software artifacts – for example, an operation is typically defined once and called from many different contexts.

In the attributed object graph model, each node represents a semantically meaningful structure, such as an individual requirement, a subsystem, an operation, or an operator within a logical expression. The nodes are the finest grain structures visible to the attribute computation rules. Each node is an instance of an abstract data type. The computed attributes of each node correspond to the operations of the data type. Invoking appropriate methods of the data type can derive the value of an attribute. Attribute computation rules are declarative definitions of these methods.

The semantics of attribute evaluation in the attributed object graph model is a generalization of the corresponding semantics in an ordinary attribute grammar. The two are the same when the graph is a tree. The difference shows up for inherited attributes of shared nodes: in an attribute grammar, each node can have at most one parent, but a shared node in an attributed object graph can have more than one parent.

We require the type of an inherited attribute to be a lattice. In implementation terms, the type must implement the lattice [T] interface with operations and these operations must satisfy the standard properties of a mathematical lattice.

$bottom : T$	-- least element
$lub(T,T) : T$	-- least upper bound
$le(T,T) : bool$	-- approximation ordering

The semantics of an inherited attribute A with a defining expression E is the least upper bound of the values of E in all contexts (i.e. the set of all parent nodes). In implementation terms, an attribute computation rule of the form $child.A = E(parent.A)$ can be realized with an initialization $node.A := bottom$ (for all nodes) and an incremental update step $child.A := lub(child.A, E(parent.A))$ which is enabled in the context of each parent node whenever the value of $parent.A$ changes in that context.

To make the above restriction on attribute types less burdensome, we propose a default extension of all types (a uniform subtype definition) that adds a new constant "bottom" representing an undefined value, another new constant "conflict_error" representing a conflict between two incompatible values inherited from different contexts, and the usual flat ordering on simple data types:

$$le(x,y) = (x = bottom)\ or\ (x = y)\ or\ (y = conflict\_error)$$

$$\begin{aligned} lub(x,y)\ &= if\ (x = bottom)\ or\ (x = y)\ then\ y \\ &\quad else\ if(y = bottom)\ then\ x \\ &\quad else\ conflict\_error \end{aligned}$$ --display an error diagnosis

This default can be explicitly overridden by the designer for data types where this makes sense. An example from the domain of timing constraints illustrates the idea:

TYPE DEADLINE EXTENDS INTEGER

$$bottom = MAXIMUM\_INTEGER$$
$$le(x,y) = x \geq y$$
$$lub(x,y) = MIN(x,y)$$

This corresponds to the idea that if a program meets a given deadline, then it also meets any later deadline. Thus, a component that inherits deadlines of 100ms, 75ms, and 120ms from three different requirement documents is subject to a design constraint to execute within 75ms (since $lub(lub(100,75),120) = 75$ for the deadline type defined above).

To ensure the high confidence of SoES, it is important to keep timing properties consistent during the whole development processes. This means the information related to timing properties needs to be consistently identified in documents belonging to different development phases. In the next section, we will use timing properties as an example to illustrate the application of the proposed attributed object graph model to the problem of maintaining document consistency.

IV. ATTRIBUTED COMPUTATIONS FOR DOCUMENT MANAGEMENT

The attributed object graph model was designed to realize documentation checks and transformations that support high confidence SoES development. These computations are used to (a) calculate the attributes from the information in the documentation repository, (b) transform the information from one development phase to another, (c) analyze the consistency between the information transformed between development phases, description languages and information views, and (d) extract subsets of documents needed for particular purposes. The declarations of these computations form a set of attribute computation rules.

In the development process, the documentation generated in early development phases is taken as input for the next phases and guides the development activities in that phase to generate the output documentation. To ensure the quality of the end product, it is important to keep selected non-functional properties needed for high confidence visible and consistent during the whole development process. These high-confidence properties should be kept consistent between the documentation generated in the early phase and that generated in the next phase. Although the format of this kind of "key information" may be different between two development phases, this information of later phase should imply that of the earlier. For example, in the requirement phase, requirement documentation may include information describing a customer request for deriving the computation result within containing constraints, then in the design phase, the design documentation should include information with the same implication, such as information related to the deadline, period and maximum execution time.

In this paper, we use timing properties transformation between requirement phase and design phase as the example to illustrate the application of attribute computation rules. Suppose that the requirements specification includes a maximum response time (MRT) constraint for a given service S and that at the architectural level, S is realized by a software component C. The maximum response time appears at the requirements level because it is directly visible to the system stakeholders and is of vital concern to them, since late control signals can have catastrophic consequences.

At the design level, this constraint is transformed into lower level constraints on the period and maximum execution time (MET) of a periodic software process. If the documentation element S in the requirements document is a parent node of the documentation element C in the design document, the design rule that ensures consistency of the two documents with respect to this issue can be expressed by the following simple attribute computation rules: (MRT is an attribute of S; timing_check, period, MET and diagnostic are four attributes of C.)

$$C.timing\_check = (C.period + C.MET \leq S.MRT)$$
$$C.diagnostic = Unless\ (C.timing\_check,\ error\_message)$$

-- Unless (C, M) displays M if $C = false$ and does nothing otherwise

The rationale for this rule is that the worst case occurs when a request arrives just after the request stream has been polled. In this case, the transaction will start processing one period later, and the software can take up to the maximum execution time after the transaction starts to produce the result. This simplified example assumes that all processing is done locally, so that we do not have to account for any latency in the communications link between the machine running the component C and the machine running the consumer process waiting for the output of C.

A mature documentation repository will actively check many different generic design rules like the one illustrated in this simple example. The rule base will gradually grow as processes are improved and constraints related to high confidence attributes are gradually formalized.

V. CONCLUSIONS AND FUTURE WORK

In recent years, complex embedded systems, known as systems of embedded systems (SoES), have been widely used in many fields such as flight control and avionics, industrial process control, weapon system control and nuclear plant control. The high complexity of SoES forces them to confront many software development challenges, such as difficulty ensuring software quality, difficulty supporting software evolution and difficulty supporting communication among different stakeholders. Much research on individual embedded system development has demonstrated that documentation plays an important role in development process and provides a promising way to cope with these challenges. In our research, we are developing a documentation driven development method for SoES. This is a new approach to documentation that can enhance integration of computer aided software development methods, encompass the entire life cycle, support system evolution and improve communication with system stakeholders. This effort enables documentation to provide more effective support for complex SoES development.

Furthermore, keeping information transformation consistent between successive development phases is an important issue in the proposed approach. It is critical for ensuring high confidence in the end product. In this paper, we investigate a specific method to perform information consistency checking in documentation driven development of SoES. We present an attributed object graph model to describe the semantics of document elements. Based on this model, we show how attribute computation rules can be used to analyze consistency between the key information such as timing properties transformed from one development phase to another.

However, further work still needs to be done in order to improve capability of documentation to efficiently support complex embedded system development. For example, a better language for defining attribute computations and an optimized evaluation engine that can handle the generalized attribute semantics proposed here should be designed.

VI. REFERENCE

[1] M. Maier, "Architecting Principles for Systems-of-System", *Technical Report*, *http://www.infoed.com/Open/PAPERS/systems.htm.*

[2] http://www.nist.gov/itl/div897/pubs/fips105.pdf

[3] L. Goldin, D. Berry, "AbstFinder: A Prototype Abstraction Finder for Natural Language Text for Use in Requirement Elicitation", *Automated Software Engineering*, No.4, 1997, pp.375-412.

[4] E. Clarke, E. Emerson and A. Sistla, "Automatic Verification of finite state concurrent systems using temporal logic specification",*http://citeseer.nj.nec.com/clarke93verification.html.*

[5] M. Dwyer, J. Hatcliff, and G. Avrunin, "Software Model Checking for Embedded Systems", *www.cis.ksu.edu/~dwyer/projects/HCES-May-01-1.ppt.*

[6] D. Garlan, "Model Checking Publish-Subscribe Software Architectures", *Presentation at ARO Kickoff Meeting*, University of Pennsylvania, Philadelphia, PA, May 24 - 25, 2001, www.cs.cmu.edu/~svc/talks/ppt/garlan.ppt

[7] J. Wing, "Scenario Graph Generation and MDP-Based Analysis", *Presentation at ARO Kickoff Meeting*, University of Pennsylvania, Philadelphia, PA, May 24 - 25, 2001, http://www-2.cs.cmu.edu/~svc/talks/html/wing_files/frame.htm.

[8] J. French, J. Knight and A. Powell, "Applying Hypertext Structures to Software Documentation", *www.cs.virginia.edu/~cyberia/papers/IPM97.pdf.*

[9] http://www.comlab.ox.ac.uk/archive/formal-methods/

[10] G. Booch, J. Rumbaugh and I. Jacobson, "The Unified Modeling Language User Guide", Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, 1999.

[11] Object Modeling Group, Inc., "Unified Modeling Language Specification, version 1.3", June 1999.

[12] M. Kande, V. Crettaz, A. Strohmeier and S. Sendall, "Bridging the Gap between IEEE 1471, Architecture Description Languages and UML", *http://icwww.epfl.ch/publications/documents*

[13] N. Mehta, N. Medvidovic, "Towards a Taxonomy of software Connectors", *in 22th International Conference on Software Engineering*, Limerick Ireland, 2000, sunset.usc.edu/classes/cs599_2000/Conn-ICSE2000.pdf.

[14] Luqi, V. Berzins and R. Yeh, "A Prototyping Language for Real-Time Software", *IEEE Transaction on Software Engineering*, Vol.14, No.10, 1988, pp.1409-1423.

[15] Luqi, R. Steigerwald, G. Hughes and V. Berzins, "CAPS as a Requirement Engineering Tool". *in Tri-Ada'91 International Conference*, San Jose, USA, Oct 22-25, 1991, pp. 75-83.

[16] R. Alur, T. Dang, J. Esposito, Y. Hur, F. Ivancic, V. Kumar, I. Lee, *et al.*, "Hierarchical Modeling and Analysis of Embedded Systems", *in IEEE*, Vol. 91, No 1, January, 2003, pp. 11-28.

[17] R. Alur, "Model-based Design of Embedded Software", *Presentation at Vanderbilt Workshop*, Vanderbilt University, Nashville, TN, December 13-14, 2001, www.hpcc.gov/iwg/sdp/vanderbilt/agenda_presentations/alur.pdf.

[18] O. Sokolsky, A. Philippou, I. Lee and K. Christou, "Modeling and Analysis of Power-Aware Systems", *in 9th International Conference on Tools and Algorithms for Construction and Analysis Systems (TACAS03)*, Warsaw, Poland, April 7-11, 2003, pp.409-425.

[19] L. Wills, S. Sander, S. Kannan, A. Kahn, J. Prasad, and D. Schrage, "An Open Control Platform for Reconfigurable, Distributed, Hierarchical Control Systems", *in 2000 Digital Avionics Systems Conference*, Philadelphia, PA, October,2000,http://controls.ae.gatech.edu/papers/kannan_dasc_00.pdf.

[20] P. Devanbu, P. Selfridge, R. Branchman and B. Ballard, "LaSSIE: a Knowledge-based Software Information System", *in IEEE 12th International Conference on software Engineering*, 1990, pp.249-261.

[21] C. Paris, K. Linden, "Building Knowledge Bases for the Generation of Software Documentation", *http://acl.ldc.upenn.edu/C/C96/C96-2124.pdf.*

[22] Luqi, L. Zhang, " Documentation Driven Agile Development for Systems of Embedded Systems", *Submitted to Monterey Workshop 2003.*

[23] G. Hedin, "Reference Attributed Grammars", *in Second workshop on Attribute Grammars and their Applications (WAGA99)*, March 1999, pp. 158-172, http://wwwrocq.inria.fr/oscar/www/fnc2/WAGA99/proceedings/hedin/hedin2.pdf

[24] D. Parigot, G. Roussel, E. Duris and M. Jourdan, "Attribute Grammars: a Declarative Functional Language", *http://www.inria.fr/rrrt/rr-2662.html*, 1995.

[25] R. Herndon, V. Berzins, " The Realizable Benefits of a Language Prototyping Language", *IEEE Transaction on Software Engineering*, Vol. 14, No. 6, June 1988, pp. 803-809.

[26] U. Kastens, "Modularity and reusability in Attribute Grammars",*http://citeseer.nj.nec.com/kastens92modularity.html*, 1992.

[27] D. Knuth, "Semantics of Context-free Language", *Journal of Mathematical System Theory*, Vol. 2, No. 2, June, 1968, pp.127-145.

[28] Ibrahim, "A Model and Decision Support Mechanism for Software Requirement Engineering", *Naval Postgraduate School, Ph.D. Dissertation*, September 1996.

[29] M. Harn, V. Berzins and Luqi, "A Dependency Computing Model for Software Evolution", *in 11th International Conference on Software Engineering and Knowledge Engineering*, June 17-19, 1999, Kaiserslautern, Germany.

[30] M. Harn, V. Berzins and Luqi, "Software Evolution Process via a Relational Hypergraph Model", *in IEEE/IEEJ/JSAI International Conference on Intelligent Transportation Systems*, Tokyo, Japan, October 5-8, 1999.

[31] S. Badr, V. Berzins, "A Software Evolution Control Model", *in Monterey Workshop 94, Monterey*, CA, September 7-9, 1994, pp. 160-171.

[32] S. Badr, " A Model and Algorithms for A Software Evolution Control System", *Naval Postgraduate School, Ph.D. Dissertation*, December 1993.

[33] M. Harn, "Computer Aided Software Evolution based on Inferred Dependencies", *Naval Postgraduate School, Ph.D. Dissertation*, December 1999.