*International Journal of Engineering & Technology*, 7 (1.1) (2018) 74-77

International Journal of Engineering & Technology

*Website: www.sciencepubco.com/index.php/IJET*

***Research paper***

# Proactive model based testing and evaluation for component-based systems

**A. Surendar [1*], M. Kavitha [2], V. Saravanakumar [3]**

*[1]Assistant Professor, Vignan's University, Guntur.*
*[2]R&D Head, Synthesishub, Salem.*
*[3]R&D Head, Bonfring Technology Solutions, Coimbatore.*
**Corresponding author E-mail: surendararavindhan@gmail.com*

**Abstract**

Embedded software systems are getting more and more complex. The demand for new features and functions led to an increasing complexity in the design and development of these systems. There are frequent reports in the media about software systems crashing and damages occurring due to software errors. One reason for this is that there are many software testing methods and techniques but they are often non-practical and difficult to use. The aim of the study was to improve existing testing methods and their practicality especially from the integrator viewpoint. Component-based system development, components of different granularities must be tested. Furthermore, an optimization approach based on simulated annealing is presented which is used to derive an integration order with respect to the proposed parameters in a powerful and reliable manner. The paper discusses explicit properties and the requirements that are to be verified, imposed upon software-intensive systems by their environment and by their users.

***Keywords:*** *Embedded Systems, Component Based Systems.*

## 1. Introduction

The aim of the study was to improve existing testing methods and their practicality especially from the integrator viewpoint. The objective was to improve interoperability between applications, and familiarize software companies and their customers with conformance testing. The integration of software components is an important aspect of embedded system development. Component-based technology has been extensively used for many years to develop software systems in desktop environments, office applications, and web-based distributed application. The advantages are achieved by facilitating the reuse of components and their architecture, raising the level of abstraction for software construction, and sharing standardized services. The use of components has partly shifted the designers' attention from algorithms to the interaction of algorithms (and their collections). This is because any component may comprise more than one algorithm, whereas precise description of algorithms used in a component and component's inner structure are very seldom known to the designer.

## 2. Software quality

We mean by a software developer a software organization that develops software for the use of end users. An integrator acquires software parts from the developers and also develops own components. The integrator integrates components into a system and tests it as a whole before delivering it to a customer. A software customer buys software from developers or integrators and carries out acceptance tests. A software integration strategy is needed to provide software testers a guideline to perform software integration testing activities in a rational way. It usually describes an order in which components are integrated and tested. The search for efficient information representation and encapsulation methods that would lead to natural software structuring, has been a driving force for software engineering. The evolution of information encapsulation methods started from modular programming, followed by object-oriented programming and design, and eventually reached the era of component-based software. A component is usually, but not necessarily always, a collection of objects that has limited autonomy, i.e. a component can exist, and to certain extent operate in a stand-alone mode. For its full-scale operation a component usually requires a specific supporting infrastructure.

## 3. Oriental software engineering

The object-oriented approach has different characteristics when compared with procedural programs, such as inheritance, polymorphism, message passing, state-based behavior, encapsulation, and information hiding. Furthermore, the execution order of the methods is not necessarily predefined, and the structure of the object-oriented programs is different from that of procedural programs. The advantages of a component-based approach are the possibility to master development and deployment complexity, modularity, decreased time to market, the quality and reusability of software and its components, the composed services of components, and the scalability and adaptability of software systems. Furthermore, software suppliers can specialize in their strategic competitive edge and buy other properties as ready-made COTS (commercial-off-theshelf) components. Among many other challenges in component-based software development, components must be put together to form the entire software system. Therefore components must be

integrated which can be illustrated as a mechanical process of wiring components together. In software integration is defined as the process of combining software components, hardware components, or both into an overall system. To interact with each other, the interfaces of components are connected by dependencies. If a component C2 uses one or more service(s) of another component C1, the formulation C2"depends" on C1 is used . The testing of these dependencies, called integration testing, insures the consistency of component interfaces and whether the components pass data and control correctly, which results in successful integration of dependent components. In other words, integration testing ensures the correct interaction between already tested components. Software integration and integration testing are often used synonymous and are not distinguishable in literature.

## 4. Concepts of computing models

Ubiquitous (and pervasive) computing is based on the expansion of the principles applied in real-time systems and plug-and-play experiments. Computationally new concepts have emerged from the domain of ubiquitous computing in relation with autonomic computing. Considering the time issue, each autonomous component may have its own time counting system and each of those time counting systems may apply its own metrics. Strictly speaking, the time instants and intervals defined in different time counting systems (time models) can only be compared within known uncertainty limits. Hence, one time dimension for the whole computing system – that so far has been the conventional approach in computer science and software engineering – cannot solve the time awareness problem. A component-based system is not monolithic: it contains components of different granularities (e.g. lowest level components, business components, and component based systems), which are integrated with other components and into legacy systems with interfaces. In such situations, testing and documentation are even more important than in conventional software projects with monolithic applications. When moving from legacy systems to component-based systems, interfaces and interface testing are needed. The components do not have to know each other's implementation, only the content of the interfaces, i.e. syntax, semantics, and instructions for using the interface.

## 5. Taxonomy of computations

For systematic progress in developing the time-aware interaction-centered model it would be desirable to categorise the variety of models of computation according to their characteristic features. The earlier used dimensions of the feature spaces applied for taxonomy could not explicitly emphasise the specific properties of context-aware, proactive computing systems. Therefore we suggest the following three dimensional approximation of the feature space–action, interaction, and time-awareness. Further we demonstrate that this feature space clearly distinguishes the conventional models of computation based on the Church-Turing algorithm theory, models of interactive computation, and models for context-aware, interactive computing. Taxonomy in Figure 3 fixes relative positions of conventional models for algorithmic computing, models for interactive computing, and models for timeaware interactive computing. On such a generic level the taxonomy is of little practical use, but if the same taxonomy be used to position more specific products–e.g. Persistent Turing Machines, Abstract State Machines, π-calculus, the Q-model–some useful hints might be extracted for guiding the further research into models for time-aware, proactive, interactive computing. The suggested feature space stems from the expected properties and requirements of the rapidly spreading new classes of computer applications–such as ubiquitous computing that includes autonomic and proactive components, computing systems with dynamic ad hoc architecture, multi-agent systems, time- and location aware computing systems etc.

## 6. Software integration

In order to evaluate the proposed parameters, two reference systems are introduced. These real-life examples are taken from the automotive industry. The first one represents an embedded data logger for battery management and consists of 16 components and 23 dependencies. The most common used criteria for evaluating an integration order is called test effort and describes the effort for creating stubs needed during integration testing. There are several approaches presented in literature to compute the test effort. Code-based testing techniques (or white-box testing techniques) study the source code and describe the code coverage: for example, whether all the statements/branches of the program are executed at least once. They do not tell whether the program is doing what the requirement specification says it is supposed to do. Code-based testing uses either control-flow criteria or data-flow criteria for test case generation. Control-flow-based testing techniques select test cases on the basis of the program's control flow. Examples of control-flow-based testing techniques are sentence coverage, branch coverage, condition coverage, and path coverage. Dataflow-based testing techniques explore the events related to the status of data objects (variables) during the program's execution. The essential events are the assignments of value and the uses of value, i.e. where the variables are defined and where they are used. Examples of data-flow testing techniques are all-definitions, all-c-uses, all-puses, and all-du-paths (c means computation, p predicate, and du definition-use pair). However, these techniques are quite theoretical and complex to use in practice. Furthermore, the customer and the integrator cannot usually use any of the code-based testing techniques because the source code is not necessarily available and even if it were there would be an enormous amount of code lines to go through. A component-based system is not monolithic: it contains components of different granularities (e.g. lowest level components, business components, and component based systems), which are integrated with other components and into legacy systems with interfaces. In such situations, testing and documentation are even more important than in conventional software projects with monolithic applications.

## 7. Simulated annealing

Parameters and the corresponding metrics help system integrators to evaluate a certain integration order; they will not provide an order which meets the corresponding requirements. To overcome this restriction, a novel approach for deriving an integration order is presented. The approach described in the following section optimizes an integration order with respect to a single parameter as well as combinations of them. Since deriving an integration order is a NP-hard problem, a heuristic optimization approach based on simulated annealing (SA) was used. The method of simulated annealing is a suitable solution for large scale optimization problems. When adapted efficiently to optimization problems, simulated annealing is often characterized by fast convergence and ease of implementation for real-world problems, Simulated annealing is based on the analogy between finding a global minimum of a cost function for a combinatorial optimization problem and the slow cooling down of metal to its minimum energy state. The configuration represents a solution, including the initial solution, of the problem. The components are numbered i=0...C−1, where C represents the number of components of the software system. The configuration spaces denotes all possible permutations of C. Therefore a configuration is a permutation of the number 0...C −1, interpreted as the order in which components are integrated. The initial solution is selected randomly. Rearrangement describes the mechanism for neighbor generation. An essential requirement for simulated annealing is

that the rearrangement mechanism provides a move from the initial state to the optimal state in a sufficiently small number of steps. Based on the configuration definition, a rearrangement function that swaps two arbitrary components can get from any state (integration order) to any other state in (C − 1) steps.
The results indicate that the proposed approach provides at least comparable results in comparison to the graph-based solutions in case of specific stubs. In case of realistic stubs, which denote the number of components to be stubbed, the simulated annealing approach obtains significantly better results on both reference systems.

## 8. Work model

Although a great deal of research has addressed the overall process of component-based software engineering (CBSE) on requirements engineering, design and evaluations, we do not have as much research on testing CBSE. Testing CBS is a challenging area of research. Existing knowledge in this field shows that CBSE introduces new problems for testing and maintaining software systems and we need new ways to validate software components, especially when they are integrated into new environments. There are a number of component-based testing methods and techniques which have different paradigms, characteristics and perspectives. The technique makes use of complete information from components for which source code is available and partial information from those for which source code is not available. Their approach separated the testing of the component-provider from the testing of the component-user, so it presented two different techniques for each category. It models the behavior of each component, specifies component interactions, and annotates the state machines with test requirements to construct a global behavioral model of the composed state charts. Then, test cases are automatically derived from the annotated state charts and global behavioral model, and executed to verify component conformance behavior. Their results show that, in most cases, state-based testing techniques are not likely to be sufficient by themselves to detect most of the faults present in the code, and they need to be complemented with other testing methods. The above approaches use only one kind of behavioral UML model for test generation, either sequence diagrams or state machines. The approach in this dissertation is novel in that it combines the information from component level UML sequence diagrams and state charts to derive a graph-based test model for the purposes of test input generation. They presented a test model that depicts a generic infrastructure of component based systems and identified key test elements. A Component Interaction Graph is generated from the implementation, in which the interactions and the dependence relationships among components are illustrated. Test adequacy criteria were developed to cover context dependence relationship and content dependence relationship. While Wu's test elements and test criteria are useful to test component-based software, their work is in the stage of approach development. This paper does not discuss and give 20 practical ways on how to use their approach to generate actual test cases for component based testing. Their test model mainly illustrates the context/content-dependence relationships defined in the paper. Additional work is required to effectively drive test generation from the test model. In addition, the authors made several assumptions in their work, including: (i) assuming that each individual component has been adequately tested by the component providers when testing component-based software; (ii) assuming that each interface only includes one operation, and the references to the interfaces and to the operation are identical. These assumptions imply that their work considers only some simplified situations, which could have limitations in applying their approach to actual component-based testing practice. From the above survey, we note that different kinds of UML diagrams have been used for software testing from different perspectives. UML state charts have been widely used to test the state-based behavior of software. Similarly, UML interaction diagrams have been used for integration testing. However, existing approaches do not focus on exercising the composition behavior of interacting components. More specifically, none of the above papers discuss testing by integrating UML interaction and state chart diagrams to uncover component interaction faults. The goal is to check whether an extracted model satisfies a certain specification. My test method, in contrast, defines input data to the object program and observes the reactions of the program. The goal of my testing is to find cases where the software reactions do not meet its expected results. There has also been research on component-based software engineering for embedded systems such as [26], which focused on embedded software. There has been work on using informal specifications to test embedded systems focusing on the application layer. A common communication protocol provides support for implementing reusable test components. Especially in the case of embedded systems, a good host test environment enables efficient software testing. When this environment matches the target system as much as possible, efficient host testing is possible. One way to support testing is to use an operating system that is supported on both the target hardware and in a host-testing environment, as simulated on a desktop. Including support for test automation as a first-class feature allows more effective analysis of the system, including analysis of long running tests and deployed systems, and enables efficient field-testing. Effectively implementing this requires possibilities for dynamic configuration of test functionality during execution. Abstracting test cases from the implementation minimizes the effects of internal system changes to the 24 test cases. This mostly applies at the system testing level, as in earlier testing phases it is often necessary to observe more detailed properties of the system.

## 9. Results

The described testing procedure has been conducted for all software components of the Safety Platform that have inputs controllable and outputs observable from within the application program. Special and hardware dependent components, e.g. drivers for digital inputs and outputs, have been either manually tested or the testing has been performed indirectly through test cases of the respective hardware unit.

**Table 1**: Component Testing

| Component | Variant | Defect | Analysis | Solution |
|---|---|---|---|---|
| CHKINC | CHKINC | Incorrect LDIFF for constant INPUT (255 instead of -1). | LDIFF is auxilary output, not used in applications. | Simulink component modified. |
| Copy | COPY | Defines logical output instead of arithmetical. | Error in graphical library. | Corrected. |
| CRCCALC | all | ENABLE input not functional. | Input not used. | Simulink component modified. |
| GENINC | GENINC | OV output is never active. | A bug in element, but OV output is not used. | Simulink component modified. |
| Summation | DIFF | Negative overflow to -32767 instead to -32768. | A bug in element implementation. | Corrected. |

In practice, mostly "hill climbing" methods are used, as advanced algorithms can be hard to implement due to computing requirements, while other types do not operate with adequate precision for modern systems, [136]. Tested algorithm is a "hill climbing" variation known as Incremental Conductance algorithm, [137]. The algorithm is based on assessment of the slope of powervoltage curve of the photovoltaic panel.

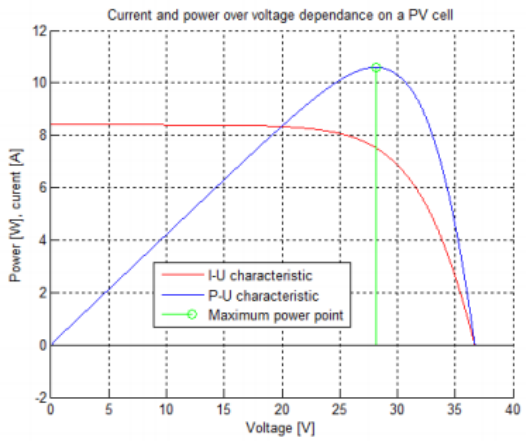


**Figure 1**: Model response

Execution time of the MPPT algorithm has been measured during open-loop real-time testing and is plotted against algorithm's output. Execution time jumps to 408 CPU cycles at start of execution, oscillates between 395 and 404 cycles during transient, rises to 406 cycles at end of transient and finally stabilizes on 395 cycles in steady state. This kind of measurement can be a starting point for in-depth analysis of SUT real-time behaviour.

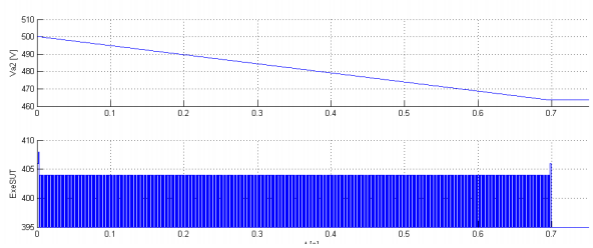

**Figure 2**: Execution time of the model based approach

## 10. Conclusion

The paper focuses on properties, development methods, analysis methods, and tools for software-intensive systems directly interacting with their environment. Many such systems are built from autonomous components that may exhibit proactive behaviour. Software-intensive systems differ from the other engineering systems in that they are clearly more capable for explicit proactive behaviour and rely on dynamic control structure more often as compared to the non-software-intensive systems in the artificial world. This paper states that applications of software-intensive systems require properties that cannot be studied by conventional mainstream methods of computer science, and suggests that a new time-aware model of interactive computation is to be developed. In order to meet this challenge, component-based architectures where introduced to automotive embedded systems. Despite the usage of eg. software product lines, a significant portion of new components must be integrated in each development step. In order to derive an integration order with respects to the proposed parameters an optimization approach based on simulated annealing was developed. In addition to minimize the singe objectives test effort and schedule effort, reasonable combinations were evaluated. It has been shown that minimizing the test effort and minimizing test complexity, which are contrary goals, can be performed by the proposed approach in an sophisticated and reliable manner. Also adding the schedule effort as objective yields favorable results. Optimizing the stub complexity and the schedule effort, which are independent goals, is also possible with good results.

## References

[1] Panchumarthi GP & Surendar A, "A review article on Fin-FET based self-checking full adders", *Journal of Advanced Research in Dynamical and Control Systems*, Vol.9, No.4, (2017).

[2] Garlan D, "Formal modeling and analysis of software architecture: Components, connectors, and events", *Formal Methods for Software Architectures*, (2003), pp.1-24.

[3] Marijan S, "Control electronics of TMK2200 type tramcar for the City of Zagreb", *Proc. International Symposium on Industrial Electronics, ISIE*, (2005), pp.1617–1622

[4] Fredriksson J & Land R, "Reusable component analysis for component-based embedded real-time systems", *29th International Conference on Information Technology Interfaces*, (2007), pp.615–620.

[5] Gallagher L & Offutt J, "Test Sequence Generation For Integration Testing Of Component Software 1", *The Computer Journal*, Vol.52, No.5, (2007), pp.514-529.

[6] Ganesan S, Alladi V, Wei J & Alladi K, "Designing embedded real-time systems (ERTS) with model driven architecture (MDA)", *SAE Technical Paper*, (2004).

[7] Lamport L, "The Temporal Logic of Actions", *ACM Transactions on Programming Languages and Systems*, Vol.16, (1994), pp.872–923.

[8] Markey N, Larsen KG & Bouyer P, "Model Checking One-clock Priced Timed Automata", *Logical Methods in Computer Science*, Vol.4, No.2:9, (2008), pp.1–28.

[9] Vimalkumar MN, Helenprabha K & Surendar A, "Classification of mammographic image abnormalities based on emo and LS-SVM techniques", *Research Journal of Biotechnology*, Vol.12, No.1, (2017), pp.35-40.

[10] Manju K, Sabeenian RS & Surendar A, "A review on optic disc and cup segmentation", *Biomedical and Pharmacology Journal*, Vol.10, No.1, (2017), pp.373-379.