Addressing Software Engineering Issues in Real-Time Software Development Environment

[1]Anujeonye Nneamaka Christiana , [2]Okonta, Okechukwu Emmanuel, [3]Wemembu, Uchenna Raphael , [4]Imiere, E. E. , [5]Aghware Fidelis

[1,2]Department of Computer Science, Federal College of Education (Tech) Asaba.
[2] Department of Mathematics Federal College of Education (Tech) Asaba.
[3] Department of Mathematics, College of Education Agbor.
[4]Department of Physics Federal College of Education (T) Asaba
[5]Department of Computer Science College of Education Agbor.

Abstract

Real-time systems are normally deployed in a wide range of applications such as transportation systems, manufacturing process, process control, military, space exploration, and telecommunications. These systems must satisfy not only logical functional requirements but also physical properties such as timeliness, Quality of Service and reliability. The cross-cutting behaviours imposed by these functional properties and dependencies on operational characteristics such as hardware, Operating System and firmware platforms that are used; have traditionally led to hard-to-code, hard-to-understand and hard-to-change software that are engineered. In this research paper we have identified few software engineering issues in the development of real time systems and provided brief description of each of those issues and strived to make serious effort to proffer creditable and functional solutions.

Key Words: Real time systems, logical functional requirements, Quality of Service, Reliability, software engineering issues

Introduction

Real-time computing is an enabling technology for many important application areas, including process control, nuclear power plants, agile manufacturing, intelligent vehicle highway systems, air-traffic control, telecommunications, multimedia, real-time simulation, virtual reality, medical applications, and military applications. In almost all safety-critical systems and many embedded computer systems are visible real-time systems. Further, real-time technology is becoming increasingly important and pervasive, e.g., more and more infrastructure of the world depends on it. Strategic directions for research in real-time computing involve addressing new types of real-time systems including open real-time systems, globally distributed real-time, and multimedia systems. For each of these, research is required in the areas of system evolution, the actual software engineering, the science of performance guarantees, reliability and formal verification, general system issues, programming languages, and education. Economic and safety considerations, as well as the special problems that timing constraints cause, must be taken into account in the solutions. [2]

In this research paper, we will be surveying some research activities carried out in the field of Software Engineering with relation to Real-Time systems. A hard real-time computer system is required to produce the intended result before a specified point of physical time, the deadline. This point of time is determined by the application the computer system is intended to service. The controlling real-time software must be designed to generate the correct behaviour of the computer both in the value domain and in the temporal domain to meet these application requirements. Since the temporal behaviour of the software depends on the performance of the computer hardware, software engineering for real-time systems must take into consideration the architectures and capabilities of the available computer hardware. It follows that the software design methods and architectures of real-time systems will be strongly influenced by the given hardware environment and consideration. [1]

These are some of the software engineering issues we identified in the development of real-time systems:

- Requirements Analysis
- Re-engineering
- Validation

Requirements Analysis

All engineering is about how to produce products in a disciplined process. In general, a **process** defines *who* is doing *what*, *when* and *how* to reach a certain goal. A process to build a software product or to enhance an existing one is called a **software development process**. A software development process is thus often described in terms of a set of activities needed to transform a user's *requirements* into a software system.

The client's requirements define the goal of the software development. They are prepared by the client (sometime with the help from a software engineer) to set out the services that the system is expected to provide, i.e. *functional requirements*. The functional requirements should state *what* the system should do rather than *how it is done*. Apart from functional requirements, a client may also have non-functional constraints that he or she would like to place on the system, such as the required response time or the use of a specific language standard. We must bear in mind the following facts which make the requirement capture and analysis very difficult:

- The requirements are often incomplete.
- The client's requirements are usually described in terms of concepts, objects and terminology that may not be directly understandable to software engineers.
- The client's requirements are usually unstructured and they are not rigorous, with repetitions, redundancy, vagueness, and inconsistency.
- The requirements may not be feasible. Therefore, any development process must start with the activities of capturing and analyzing the client's requirements. These activities and the associated results form the first *phase* (or sub-process) of the process called *requirement analysis*. The purpose of the requirement capture analysis is to direct the development towards the right system. Its goal is to produce a document called *requirement specification*. The whole scope of requirement capture and analysis forms the so-called *requirement engineering*.

High-level design of the computerized component of a critical system is not performed in a vacuum but strongly depends on models and assumptions regarding, besides device itself, the environment, the sensors and the actuators.

Therefore, writing the design specifications, from which the development of the device can start, is not the first activity of the development process, but must be the result of a preliminary phase whose purpose is to state, analyze, and prove the user requirements (typically stated very abstractly in terms of some safety or utility property) by modelling the system in its entirety, including the device and all the other components.

The modelling and analysis activities must be formal, to provide a support in dealing with complexity, to obtain mechanized checks for correctness, completeness, and consistency, and to certify the obtained results. This in turn requires the adoption of a formal notation, which also ensures absence of ambiguity, thus preventing misinterpretation among people, participating to system requirements analysis, who often have quite heterogeneous cultural backgrounds. To facilitate communication, discussion, and mutual understanding, the formal notation must be flexible, expressive, and high level. It must be able to model in natural way real-world entities, basic notions such as events, actions, states (i.e., properties or values of system components, possibly having non null duration), continuity or finite variability, (non) determinism, and cause-effect relations. [3]

Re-engineering

There is a growing demand for software tools that can assist in designing, analyzing, and debugging embedded real-time applications. In the literature, various techniques based on real-time scheduling theory and formal methods have been proposed and many of them are implemented into software tools. Also, a number of commercial CASE tools have been developed and widely used. While most of these tools put an emphasis on the development aspect of embedded real-time systems, in practice, a great deal of effort is put into re-engineering of already developed systems. The re-engineering of an embedded system is defined as a development task of meeting newly imposed performance requirements after its hardware and software have been fully implemented.

In the industry, a large number of new lines of products are released merely as update of older designs. During product's re-engineering cycle, developers are often faced with tasks which involve intensive hand-tuning of embedded system designs. These tasks are often very difficult to carry out since product's developments are usually under very strict cost and performance constraints. However, it is fairly obvious that such a naïve approach will fail in practice due to the excessive price of the final products. Thus, it is inevitable for the engineers to pinpoint performance bottlenecks in the old design and carefully choose only those parts that can lead to about 25% performance improvement at the least cost. Such a task of performance re-engineering will get even more difficult if the original developers have been relocated to another project, or if the original systems were developed in an ad hoc manner. Worse still, there are very few tools to aid in performing such a re-engineering task, even though engineers are under tight deadline constraints for reduced time-to-market. Performance re-engineering involves analyzing a heterogeneous distributed multiprocessor hardware platform since an embedded real-time system often consists of multiple microcontrollers, ASIC (application specific integrated circuits) chips, and electro-mechanical components. In addition, performance re-engineering possesses very distinct and inherent characteristics: (1) software and firmware code of the underlying system has been developed and well-tested; and (2) task allocation and scheduling have been already completed. [4]

Validation

At the end of the development cycle it must be decided whether a given system is safe to deploy in the intended application area. If this application area is safety critical, i.e., a failure of the computer system can result in high financial loss or even a catastrophe where human lives are endangered then this decision is difficult. Many safety critical applications demand a level of dependability that cannot be established by state of the art testing technology. Some trends in the field of validation of high dependability real-time systems are:

Process versus Product

Since it is beyond the state of the art to validate by testing that a large real-time system is free of critical design errors, the validation emphasis has shifted from the analysis of the product to the analysis of the development process of the product in the past few years.

Worst Case scenario

The time specifications at the architecture design level identify the deadlines the component must meet under all specified operational conditions. During component design it must be demonstrated, that these deadline will never be missed. A necessary prerequisite for this temporal validation is knowledge about a tight upper bound of the worst case execution time of all time-critical process inside a component.

Simulation

Large real-time systems require a closed loop simulation in the laboratory to demonstrate that the system provides the intended services.

Formal Verification

A safety case is the accumulation of evidence from different sources that establishes the rational basis for the decision that a safety critical complete system is safe to deploy. The formal analysis of critical algorithms that are used in the system can form a convincing argument in the safety case. [1]

Composing Modules with Synchronization and Real-Time Constraints Using Category Theory

Nowadays, complex real-time, embedded software systems are typically being composed out of reusable and mostly deployable components. The authors are aware of the paper presented by Varma and Sinha which presents a formal framework that utilizes the concepts of category theory to provide for a rigorous, consistent and traceable composition of modules with constraints.

The main contributions of the paper are to:

- Introduce the formal framework to facilitate the composition process.
- Define modules and their contracts for their interactions.
- Illustrate composition with constraints and its correctness using concepts of category theory.

Their paper gives an overview of component (module) composition utilizing concepts of category theory. [8]

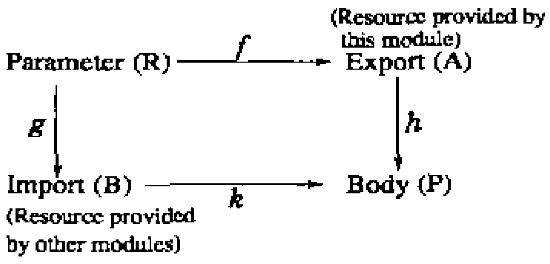

$$\begin{tikzcd}
\text{Parameter (R)} \arrow[r, "f"] \arrow[d, "g"'] & \text{Export (A)} \arrow[d, "h"] \\
\text{Import (B)} \arrow[r, "k"'] & \text{Body (P)}
\end{tikzcd}$$
(Resource provided by this module)
(Resource provided by other modules)

Figure 1: Module Interfaces

Module specifications are defined by utilizing the notion of push-out operation from category theory. Given specifications A and B, and a specification R describing syntactic and semantic requirements along with two morphisms f and g, the push-out operation gives specification R which contains A and B.

Composition of module specifications:

The composition scheme allows two modules to be interconnected via export and import interfaces. The push-out of the two modules is the resulting specification of the composed module.

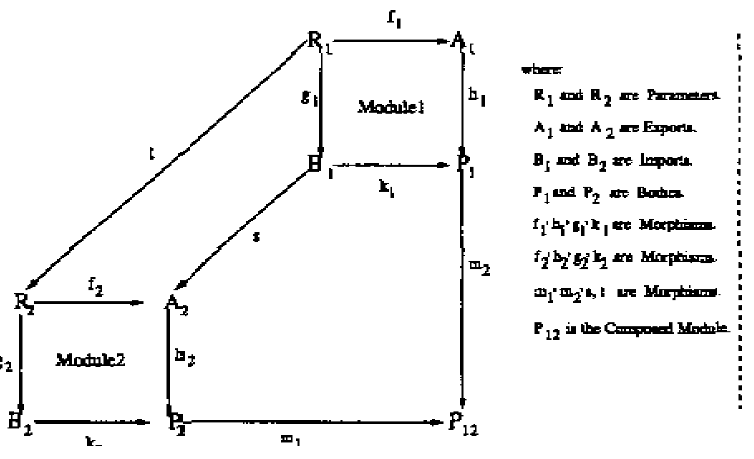

Module1
Module2
where:
R1 and R2 are Parameters.
A1 and A2 are Exports.
B1 and B2 are Imports.
P1 and P2 are Bodies.
f1, h1, g1, k1 are Morphisms.
f2, h2, g2, k2 are Morphisms.
m1, m2, s, t are Morphisms.
P12 is the Composed Module.

Figure 2: Composition of two modules

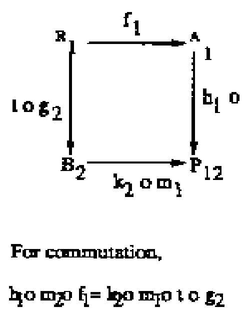


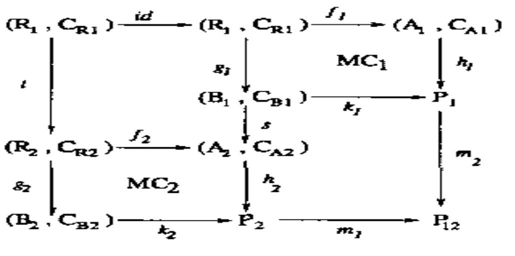
Figure 3: Composition of two modules with constraints

Figure 2 depicts the composition operation, where Module 1, M1 = (R1, A1, B1, P1) and Module 2 = (R2, A2, B2, P2). In Fig (a), Module 1 imports via specification B1 whatever Module 2 exports via specification A2. The compatibility of the parameters is governed by morphism t. In this case, the resulting composed module M12 is (R1, A1, B2, P12), where P12 is the push-out of P1 and P2 over B1. Furthermore, as the composed module commuted, i.e., its construction being proven correct, it can also be reused for subsequent composition.

Specifications with Constraints:

A module specification with constraints written as MC = (RC, AC, BC, P, f, k, g, h) consists of three specifications with constraints: (a) RC = (R, Cr), (b) AC = (A, Ca), (c) BC = (B, Ca), a specification without constraints P, and four morphisms f, k, g, h such that the basic part of M of MC given by M = (R, A, B, P, f, k, g, h) is a module specification without constraints).

Composition with Constraints:



Given two module specifications with constraints MC1 and MC2 and an interface passing morphism v from MC1 to MC2 i.e., a pair v = (s, t) of specification morphism s: (B1, CB1) -> (A2, CA2) and t: (R1, CR1) -> (R2, CR2), the composition MC12 of MC1 and MC2 via v written as MC12 = {(R1, CR1), (A1, CA1), (B2, CB2), P12}.

Union with Constraints:

Given module specifications with constraints MCj for j = 0, 1, 2 and module morphisms f1: MC0 -> MC1 and f2: MC0 -> MC2, the weak union MC3 of MC1 and MC2 via MC0 is written as MC3 = MC1 + mcoMC2. Furthermore, PC3 = PC1 + pcoPC2, RC3 = RC1 + rcoRC2, AC3 = AC1 + acoAC2, and BC3=BC1+bcoBC2.

Proposed Framework

The main objective of proposed framework is to facilitate the composition of modules with constraints. The initial step is the identification of modules (components) based on the working principles of the system. These components are then specified formally by defining sorts, operations and equations for the parameter, import and export interfaces of the component. A set of contracts or constraints for each of these components are defined along with their specification. Currently, contracts being defined include timing and synchronization constraints in the components. Other non-functional properties such as bandwidth and memory constraints can also easily be described as contracts. The composition of theses modules to result in a complete system is achieved via category theoretic operations. The final system is when verified for the correctness against a set of requirements prescribed for the system and the constraints resulting over the composition.

Concept of Contracts:

A software component can be defined as an independently deployable unit of composition with contractually specified interfaces. Internal contracts are constraints imposed on the stand-alone component. This generally deals with the initial values and constraints on the operations that can be performed by the component. External contracts are introduced as a result of inter-component interaction. The resulting constraints being imposed effect on the operation of the interacting components.

Contracts and Morphism Definitions:

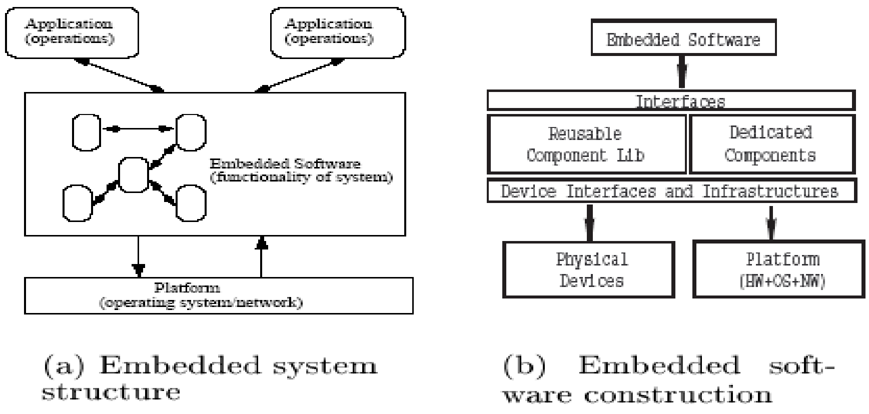
Morphisms define a rule in which two categories or components combine to form a composed category or components combine to form a composed category or a subsequently reusable component. Contracts play an important role in the morphism function definition. The morphism that combines two components is the functional implementation of the internal and external contracts that exist in each of the components. Thus, it can be summarized that morphisms are derived from the contracts that exist in each of the components.

Architecture for Embedded Software Integration using prototype Components

Behaviours of integrated software in the architecture proposed by Shige Wang and Kang G. Shin [12] are modelled as *Nested Finite State Machines* (NFSMs). The NFSM model supports compositional behaviour specifications. It further supports incremental and formal behaviour analysis. The behaviour correctness of such an integrated system can be verified using an approach similar to that in [10]. Furthermore, since a given behaviour can be implemented by different FSMs [11], different components may be selected for integration to meet different constraints while achieving the same behaviour. The behaviours specified in other models or languages can be converted to this model using translators. The integrated behaviours can then be specified in a Control Plan program for remote and runtime behaviour reconfiguration. This architecture also separates other non-functional constraints, especially timing and resource constraints, from functionality and behaviour integration so that these constraints can be analyzed and verified incrementally and as early as at design phase.

Component Structure

Components are pre-implemented software modules and treated as building blocks in integration. The integrated embedded software can be viewed as a collection of communicating reusable components. Figure 4 shows the embedded software constructed by integrating components.



Application (operations)
Application (operations)
Embedded Software (functionality of system)
Platform (operating system/network)
(a) Embedded system structure
Embedded Software
Interfaces
Reusable Component Lib
Dedicated Components
Device Interfaces and Infrastructures
Physical Devices
Platform (HW+OS+NW)
(b) Embedded software construction

Fig. 4: Integration of embedded software

The component structure defines the required information for components to cooperate with others in a system. The software component is modelled as a set of external interfaces with registration and mapping mechanisms, communication ports, control logic driver and service protocols, as shown in Figure5.

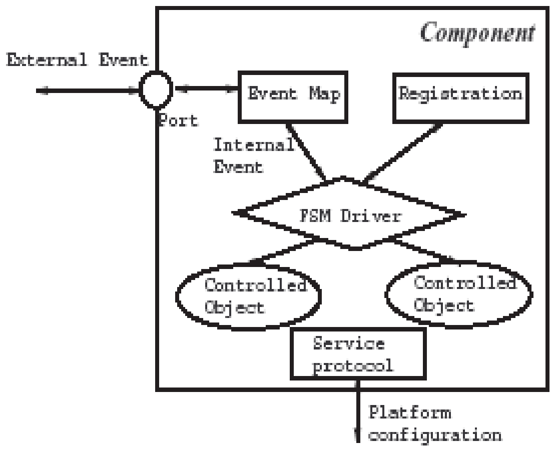


Figure 5: Reusable component structure

External interfaces:

External interfaces define the functionality of the component that can be invoked outside the component. In this model, external interfaces are represented as a set of acceptable events with designated parameters. A component with other forms of external interfaces, such as function calls, can be integrated into the system by mapping each of them to a unique event

Communication ports:

Communication ports are used to connect reusable components, i.e., they are physical interfaces of a component. Each reusable component can have one or more communication ports.

Finite State Machine driver:

The control logic driver, also called the FSM driver, is designed to separate function definitions from control logic specifications, and support control logic reconfiguration. The FSM driver can be viewed as an internal interface to access and modify the control logic, which is traditionally hard-coded in software implementation.

Service protocols:

Service protocols define the execution environment or infrastructures of a component. Example service protocols include scheduling policies, inter-process communication mechanisms and network protocols.

System Integration

Software integration includes component selection and binding, and control plan construction (both control logic and operation sequence). A runtime system can be generated by mapping the integrated software onto a platform.

Composition Model

The composition model defines how software can be integrated with given components. Since each reusable component is implemented with a set of external interfaces that uniquely define its functionality, components can be selected based on the match of their interfaces and design specifications. The integration of reusable components can be viewed as linking the components with their external interfaces. Reusable components in integrated software are organized hierarchically to support integration with different granularities, as illustrated in Figure6.

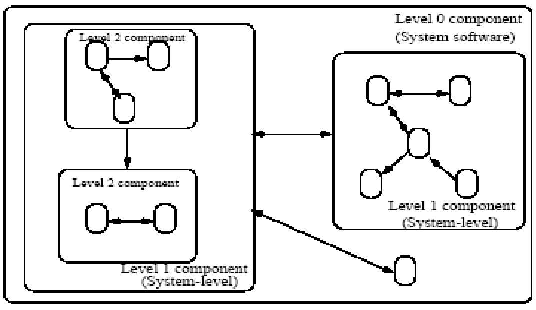


Figure 6: Hierarchical composition model

The behaviour of an integrated component can then be modelled as integration of its member component behaviours. The control logic and operation sequences of each component can be determined individually and specified in a Control Plan. The behaviour specifications can further be classified as device-dependent behaviours and device-independent behaviours. The device-independent behaviours depend only on the application level control logic, and can be reused for the same application with different devices. The device-dependent behaviours are dedicated to a device or a configuration, and can be reused for different applications with the same device.

With such a composition model, both components for low-level control such as algorithms and drivers and for high- level systems can be constructed and reused. However, additional overhead is introduced as the component level is increased, and may results in associated performance penalties due to excessive communications and code size.

Runtime System Construction

The integrated software obtained from the composition model cannot be executed directly on a platform since the composition model only deals with functionality. To obtain executable software, components have to be grouped into tasks, which are basic schedulable units in current operating systems. Each task needs to be assigned to a processor with proper scheduling parameters (e.g., scheduling policy and priority) determined by an appropriate real-time analysis. Also, communications among components should be mapped to the services supported by the platform configuration. After these pieces of information are obtained, the components can be mapped to the platform by customizing their service protocols. **Error! Reference source not found.**7 shows the mapping from functional integrated software to a runtime system with our architecture.

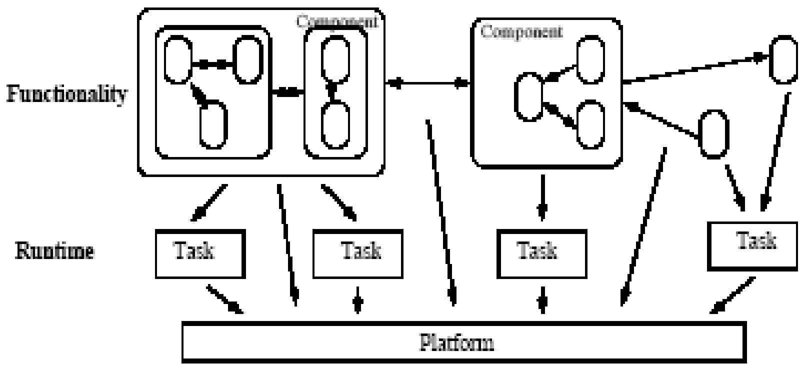


Figure 7: Runtime System Geenration from Composition

Conclusion

This paper has put forth an initial effort in development of a formal framework for composition of modules that have synchronization and real-time constraints. Category-theoretic framework is discussed to assist realizing such compositions. One of the benefits of proposed framework is that it facilitates tracing of impacts or influences of a specific constraint imposed on a modules could have on other modules over an interaction.

In addition, a component-based architecture for embedded software integration is discussed above. This architecture defines components and a composition model as well as a behaviour model. A reusable component in the architecture is modelled with a set of events as external interfaces, communication ports for connections, a control logic driver (FSM driver) for separate behaviour specification and reconfiguration, and service protocols for executing environment adaptation. Such a structure enables multi-granularity and vendor-neutral component integration, as well as behaviour reconfiguration. [12]

Lastly we discussed reusable component architecture for real-time systems. Accordingly these systems can be modelled with a set of events as external interfaces, communication ports for connections, a control logic driver for separate behaviour specification and reconfiguration, and service protocols for executing environment adaptation. The control logic of each component is specified in a state table separately from the component implementation, and can be reconfigured remotely and dynamically which also allows the verification to be done independently of implementation, and incrementally as the integration continues.

Recommendations

- The issue of reusable interface should form further research work in this area.
- The delivery method, and error detection in terms of quality assurance was also not treated because the operational specification in a real time distributed system with fail safe architecture was addressed and can form further research work in order to provide a good linking interface.

References

[1] Hermann Kopetz (2000). Software *Engineering* For Real-Time: A Roadmap. *Proceedings of The Conference On The Future Of Software Engineering*

[2] John A. Stankovic (1996) Strategic Directions in Real-Time and Embedded Systems. *ACM Computing Service, Vol. 28, No. 4.*

[3] Gargantini and Angelo Morzenti.(2001) Automated Deductive Requirements Analysis of

Critical Systems. *ACM Transactions on Software Engineering and Methodology, Vol. 10, No. 3.*

[4] Minsoo Ryu, Jungkeun Park, Kimoon Kim, Yangmin Seo, and Seongsoo Hong(1999). Performance Re-engineering of Embedded Real-Time Systems. *ACM SIGPLAN Notices , Proceedings of the ACM SIGPLAN 1999 workshop on Languages, compilers, and tools for embedded systems, Volume 34 Issue 7.*

[5] Jan Richling, Matthias Werner, Louchka and Popova-Zeugmann (2002) “Automatic Composition of Timed Petri-net Specifications for a Real-Time Architecture”

[6] C. L. Liu and James W. Layland,(1979) “Scheduling algorithms for multiprogramming in a hard-real-time environment,” *Journal of the ACM, vol. 20, no. 1.*

[7] Jan Richling (1999), “Komponierbare Echtzeitsysteme — Entwurfsmethodeund Architekturentwurf,” Tech. Rep. Informatik Bericht 127, Institut f˙ur Informatik, Humboldt-Universit¨at, Berlin, Germany.

[8] Varma, N.and Sinha, P.(2003); Composing modules with synchronization and real-time constraints using category theory. Electrical and Computer Engineering, 2003. IEEE CCECE 2003. Canadian Conference on , Volume: 2 .

[9] Kopetz, H and Suri, N.(2003) Compositional design of RT systems: a conceptual basis for specification of linking interfaces. Object-Oriented Real-Time Distributed Computing, 2003. Sixth IEEE International Symposium .

[10] R. Alur and M. Yannakakis. (1998) *Model checking of hierarchical state machines*. In Proceedings of the 6th ACM Symposium on Foundations of Software Engineering, Lake Buena Vista, FL.

[11] T. Villa (1997). Synthesis of Finite State Machines: logic Optimization. Kluwer Academic Publishers.

[12] Shige Wang and Kang G. Shin (2000). An Architecture for Embedded Software Integration Using Reusable Components. In Proceedings of the 2000 international conference on Compilers, architecture, and synthesis for embedded systems.