A Component Based Design Tool for Networked Embedded Software
Supporting Non-Functional Analysis

Ke Shi
Services Computing Technology and System Lab
School of Computer Science & Technology
Huazhong University of Science & Technology, Wuhan 430074, China
keshi@hust.edu.cn

Abstract

In this paper a new approach for building networked embedded software is presented. The approach is based on the composition of reusable components with the addition of a perspective contract principle for modeling non-functional properties. Non-functional properties are an important aspect of networked embedded software, and this is why they are modeled separately. As such, the component view presented here differs from traditional component based views, where focus is laid on the functional part. The ideas discussed in the paper have been implemented in a tool. This tool enables the construction of networked embedded software by means of components and perspective contracts. Currently, a queuing network based algorithm that considers all non-functional properties together performs a static analysis on the perspective contracts before execution of the application.

1. Introduction

Building networked embedded software from scratch is time-consuming and costly. The use of software components for constructing and tailoring these systems has promise. The introduction of the component-based software development (CBSD) paradigm into networked embedded software development offers significant benefits, namely: (i) configuration of networked embedded software for a specific application using components from the component library, thus reducing the system complexity as components can be chosen to provide exactly the functionality needed by the system; (ii) rapid development and deployment of networked embedded software as many software components, if properly designed and verified, can be reused in different applications; and (iii) evolutionary design as components can be replaced or added to the system, which is appropriate for complex networked embedded systems that require continuous hardware and software upgrades.

Although component-based software development techniques are maturing for business and desktop systems, they are less mature for networked embedded software. All these techniques have one thing in common: they focus on the functional aspects of the components and their composition. In networked embedded software, one has to consider non-functional and resource constraints when building a system. Networked embedded systems often have limited processing power, storage capacity and network bandwidth. A developer has to cope with these constraints and make sure that the software will be able to run on the constrained system.

The correct working of such a system is not only dependent on the correct functional working of the component; it is also dependent on its non-functional properties [1]. For example, using a component that consumes large amounts of internal memory in memory-constrained systems [2] is not a good idea. Often, networked embedded systems also have timing constraints on their computations. Missing a time constraint can be catastrophic (e.g. late activation of a cooling subsystem in a factory) or annoying (e.g. missing some video frames on a portable video device).

It is clear that some way is needed for the specification and checking of non-functional constraints. This will enable one to safely reuse components in a design, while being sure that the non-functional constraints will be met. In our approach, perspective contracts are used to ensure this. Our work focuses on the development of effective composition mechanisms and the associated non-functional analyses for networked embedded software.
The paper discusses some core concepts of the tool and perspective contracts in section 2. In section 3 component model and component library used in the tool is described. Section 4 discusses the composition process for building applications. The non-functional analysis method to perspective contracts check is described in section 5. In section 6, related work is discussed. Section 7 gives a conclusion.

2. The Core Concept

Our goal was to enable component-based technology with this tool to specify, compose, and validate software for networked embedded systems. Much focus has been put on non-functional properties such as memory consumption, timeliness etc. which makes the tool analyzable. To achieve this goal, perspective contract that extends the notion of contract in the CBSD area is proposed to model non-functional properties.

2.1. Design by Contract

The design by contract principle is a well-known and interesting principle. In general, a contract specifies an agreement between two or more parties about a service. This contract principle can be applied to software: since components offer services to other components, the properties of these services can be put in contracts.

There are different ways to use a contract. A first approach is to use them in a notational manner; the contract is only informing the designer about particular service properties. In this case, a contract can be seen as extra documentation. Secondly, Contracts can also by used to test an application; the contract is used to perform runtime checking of the contract properties. The contract is thus not only used for annotating the application, it is also used for monitoring the application. However, using contracts for testing does not guarantee that the contract will hold at all times and a runtime checking mechanism for these contracts has also been implemented in the special language. Finally, contracts can be submitted to an analysis process; an algorithm performs a static analysis on the contracts before execution of the application. This static analysis guarantees the correct working of the application at all times.

2.2. Perspective Contract

In networked embedded software, using contracts to model non-functional properties directly is not an appropriate method. First, non-functional properties of networked embedded software cannot be encapsulated in a component with well-defined interfaces as they crosscut the structure of the overall system, e.g., synchronization, memory optimization, power consumption, and temporal attributes. That is to say, they crosscut component boundaries. For example, changing one component may affect the end-to-end response time of many components that are working together. Second, non-functional properties are often context-sensitive, dependent on specific runtime platforms and executing tasks. Decomposing non-functional properties to specific contracts for every component in design phase is a very difficult and error-prone task for the developers and in some cases even impossible to implement.

Our solution is perspective contract, which extends the notion of contract and provides a structured and efficient way of modularizing crosscutting non-functional properties in networked embedded software. Perspective contract is independent of programming languages because language independent perspective contracts help developers handle crosscutting non-functional properties among components at the design stage, and the resultant new design can be implemented in any programming language. The definition of perspective contract includes sufficiently general, parameterized, complete English description, meaningful constraints specified, and relating to non-functional properties.

We believe perspective contract is valuable to the development of the networked embedded software. By using perspective contract a developer is encouraged to design in a functional manner and then to apply non-functional updates to the design. This separation of concerns makes design easier. Perspective contract support a widespread global change in the design by simply defining new perspective contract and applying it to your design. This prevents bugs where changes required are only made in some of the requisite places. Also implied by this advantage is that re-applying different perspective contract can be done simply and non-functional analysis can be re-run automatically. This facilitates looking at multiple competing design options and making modifications easy.

Now perspective contracts supporting by the tool include temporal property contract, memory consumption contract and network bandwidth contract. To guarantee the correct working of the application, a queuing network based algorithm performs a static analysis on the perspective contracts before execution of the application. We will detail the algorithm in section 5.
3. Component Library

Component Library contains reusable components, which is the fundamental parts to construct an execution platform closely tailored to the needs of its application(s). A component is a reusable design entity and contains the type description and implementation. The interfaces of a component are described by means of ports. Ports define a set of connection points between components to expose various roles supported by a component interface. A port has a set of messages that can be received or sent according to the protocol associated with it.

The interactions among interdependent components are defined by binding special ports of these components. A binding is a connection between two ports. Components can exchange messages via bindings. It is only possible to create a binding between ports that have the same protocol.

Our approach to model such a component is to describe component metadata separately from the implementation of the component functionality. Component metadata describes the reflective information of a component, which includes categories such as list of files used to implement the component, interface definition, configurable parameters, linking information, versioning information, performance information, resource requirement for a component to function, dependencies on other components, and other reflective information needed to non-functional analysis. These metadata do not implement application functionality per se. They are nevertheless important to the proper functioning of an application.

Our approach helps to decouple the functional aspects of a component-based application from the underlying requirements and configuration details, thereby increasing composition flexibility and systematic reuse.

Language used to define component metadata should be extensible to allow the specification of metadata that is open-ended and subject to change. The tool uses XML (extended markup language) to define and specify component metadata, which avoids the effort required to design a full-edged language and enables developers of networked embedded software to separate metadata from the component implementations, while also enabling the integration and composition of third-party code. Specifying metadata in an ad hoc manner prevents interaction with components developed using other non-compatible metadata specification mechanisms.

A component can be reused in a variety of contexts with differing requirements and runtime environments through characterizing its metadata. The metadata concept and its application are some of the key features that distinguish our tool from many other tools.

Our tool now implements three component libraries: the application library, middleware library and OS library. There is a special kind of component called pattern component in the tool library. Pattern defines an assembly abstraction to group components and characterize their metadata that describes the components present in the assembly. A pattern component is a composite component that can be used as a single component like common component. A XML pattern descriptor file describes the components that make up the pattern, how those components are partitioned, and how they are inter-connected. This mechanism enables developers constructing their own components based on special requirements and domain knowledge and easing the following develop work. Therefore, the component library is opened and extensible.

4. Configurable Composition

4.1. Configurable Composition

Composition groups components in the library to build an application running on one or more nodes to execute tasks, which communicate via the communication system connecting the nodes. Configurable composition means the components’ metadata and their interconnection could be characterized to meet the application requirements.

The whole process of configurable composition can be divided into three steps. Three model types, concept, instance and scenario models, are used respectively at each step for building applications. Concept and instance models are platform independent, and scenario model is platform dependent. Every model type has its own purpose, but the key idea is that they provide a way for decomposing a system in coherent parts.

The models consist of multiple instantiated components. Of course, the corresponding components need to be available when building these models. These components can be designed and implemented when needed, or can eventually be reused from a component library. This last aspect is very important, since reuse of existing components will shorten the development cycle.

4.2. Concept Model

A concept model is a set of component types that are being used in (a part of) the application. In fact, a concept model is a functional, static composition for
the application. An application will often consist of multiple concept models, and component types that are closely related are put in the same model.

4.3. Instance Model

An instance model is a collection of interconnected component instances. An application can consist of multiple instance models, and a component instance can be present in more than one instance model. An instance model represents the runtime situation of the application, so it represents a functional dynamic composition for the application.

This model can be compared to an UML object diagram; with the difference that an instance model is used for an exact modeling of the runtime situation (there is a one to one mapping between design and runtime) while an UML object diagram is a ‘drawing’.

4.4. Scenario Model

A scenario model consists of a collection of interconnected component instances running on special execution platform to execute required tasks, with perspective contracts attached to the whole structure. These perspective contracts are used to specify non-functional requirements on the application. It is not possible to alter the structure of the application in a scenario model. A scenario model is thus used for the non-functional dynamic composition for the application.

In this stage, the component-to-platform deployment is performed; many context sensitive parameters and configuration details are determined. Non-functional analysis could be carried out to detect whether the perspective contracts would be met. If not, the developers would have the opportunities to modify their schemes in the design phase.

5. Non-Functional Analysis

The purpose of non-function analysis is to ensure the application built from components composition meet the non-functional requirements. Almost every developer engaged in CBSE face the same challenge that individual components may function satisfactorily, but the composition of these components into higher-level applications may not meet systemic non-functional properties such as time and memory constraints. For resource limited embedded application, this challenge becomes more serious.

Many approaches have been proposed to address non-functional analysis during networked embedded software development. Most of them rely on the assumption of independent non-functional properties that can be handled individually, one at a time. Since meeting a property may potentially cause violation of other properties, we need a solution that considers all non-functional properties together. Moreover, most of them need to have global detailed knowledge about the system. This implies only runtime checking is available to determine whether the non-functional properties can be met at the system integrated testing stage. If a problem is found, the amending work will be difficult and expensive.

A queuing network approach is proposed to solve the problem. First, a multi-class queuing network model is constructed based on the information deduced from the scenario model of the application. Then, the performance, resource consumption and other measures of the application can be obtained from the queuing network model solution. Last, perspective contract checking is performed based on these measures to determine that the non-functional properties could be met or not.

5.1. Queuing Network Model

Because we are interested in modeling the whole system to get performance, resource consumption and other measures, we need to model both hardware and software.

Hardware model is used to describe the executing platform. We consider a platform with K devices where each device possesses computation, memory and communication resource. These devices are connected with some dedicated communication links. If some devices are not directly connected by dedicated communication links, there is a communication route between them. We assume that the true service discipline is round robin among all waiting tasks, so each device is modeled by a single processor sharing (PS) server in the queuing network. Each communication link can be represented by a single server in the queuing network, and we will use the term communications processor (CP) to refer to the communications servers. We assume that CPs connected with particular devices operate independently of these devices. In particular, the device service rate is independent of the number of messages being transferred by the CPs connected with it.

We will let C(i) be the index of the server that represents the device i in the queuing network and let CP(i,j) be the index of the server that represents the CP between devices i and j. We assume that messages are transmitted between devices by being divided into packets of fixed size. Furthermore, we assume that if packets from several messages are waiting for service
from a particular CP, then packets are transmitted in a round robin order among all messages waiting for the CP.

In most case, our assumption that devices and communication links process waiting tasks in round robin order and modeling them as PS servers are reasonable and enough because the system processing and communicating capacities have no direct relations with service discipline. If non-functional properties of the system can be met under round robin service discipline, we believe the system have had enough capacities to meet non-functional properties even using other service discipline. Moreover, our assumption can be justified to allow the service times at the Cs and CPs to be class dependent or define task priority to meet special service discipline

Software model describes the components and the tasks in the system. We let M be the total number of software components in the system. We assume that there is only one copy of each of the M components. Multiple instances of components can be handled by counting components more than once in this total. We assume that a component is an atomic unit and cannot be split between two devices in the system.

We suppose that there are D tasks to be run on the system. We let pᵢ be the number of instances of task i. For each task i, we let Uᵢ be the set of components used by the task. We assume that a task is executing only one component at any given time and that communications transfer time is not overlapped with computation time.

The resource consumptions such as CPU time and memory capacities for each component are specified in a platform-independent form. The platform independent form of resource consumptions can be obtained by measuring resource consumptions on a reference platform to executed reference task set and converting them to platform-independent values using techniques such as scalar [3] or virtual resource service rate [4]. The conversion function can be included as part of component’s resource consumption function F. In the model, We will use the D by M matrix EC={ECᵢ,j} and the M by N matrix RC={RCᵢ,j} to estimate the CPU resource demands from component execution. Each entry ECᵢ,j of EC with j in Uᵢ gives the total average execution time required to execute component j, whose value can be got from component metadata directly. Each entry RCᵢ,j of RC gives the rate by which the execution of component i should be adjusted when the component is executed during the running of task i at device j, whose value can be calculated through conversion function. Thus the average service time required at device j by task i for the execution of component m is given by \( \frac{ECᵢ,j}{RCᵢ,j} \).

Memory capacity also can be described at the same form. We will use the D by M matrix EM={EMᵢ,j} and the M by N matrix RM={RMᵢ,j} to estimate the memory capacity demands from component execution. Thus the average memory capacities required at device j by task i for the execution of component m is given by \( \frac{EMᵢ,j}{RMᵢ,j} \).

We use the D by M matrix C={Cᵢ,j,k}, the vectors S={Sᵢ} and L={Lᵢ} to calculate intercomponent communication costs. Entry Cᵢ,j,k gives the communications time required to transmit data from component j to component k during the execution of task i whenever component j and k are not co-resident and assuming a CP with speed normalized to 1. These entries can be estimated from the port binding definitions between component j and k. For consistency we assume Cᵢ,j,k =Cᵢ,k,j for all k, j. Entry Sᵢ,j gives the speed of the CP connecting nodes i and j. Lᵢ gives the device where component i is located in the network. Thus if j and k are components used by task i and Lᵢ is not the same as Lⱼ, then the resource demand on CP(Lᵢ ,Lⱼ) caused by execution of task i is given by \( \frac{Cᵢ,j,k}{Sᵢ,j} \).

5.2. Queuing Network Model Solution

Given a particular scenario model we can use the parameters described in the last section to calculate the parameters of a queuing network model of the system. We ignore the exact order in which the components are executed. This is possible since the performance characteristics of a queuing network are known to depend only on the total per class service time at each server in the network. In our present context, this means that only the following parameters are significant: the total mean service requirement of each task at each device in the system, and the total mean communications time required at each communication links in the network during the execution of the task.

Since the order of component execution is immaterial, we will assume that they execute in numerical order and we will define the service time of each task instance as the sum of the service time including waiting time and processing time at each device that the task instance i must visit to receive service.

Let \( T=\{tᵢ,j\} \) be the service time matrix for the queuing network. That is, let tᵢ,j be the mean service demand by customer class i at server j. (Because we have assumed the PS discipline at each server in the network, only the mean service times are significant, and the exact form of the service time distribution does not matter). We represent each task type as a distinct customer class in the queuing network. Given the module allocation vector L, tᵢ,j can be calculated according to:

\[ tᵢ,j = \sum_{k=1}^{M} \frac{Cᵢ,j,k}{Sᵢ,j} \]
Let \( c_{i,j} \) be the mean queue sizes for transferring data between \( j \) and \( k \) components at \( j \). Let \( t_{m,i,j,k} \) be the mean service time for task \( i \) at device \( j \).

\[
I_{i,j} = \sum_{i=1}^{D} \sum_{j \in U} c_{i,j} / r_{C,i,j} ,
\]

when \( j \) represents a CPU server and

\[
I_{i,j} = \sum_{i=1}^{D} \sum_{j \in U} c_{i,j,k} / \times SCP(L_i,L_j) ,
\]

when \( j \) represents a CP server.

Given the matrix \( T \), one can apply any of the standard techniques to determine performance characteristics of the queuing network. Because we were primarily interested in response time and mean queue sizes, and because the mean-value analysis (MVA) method is extremely easy to program, we chose MVA as the solution method for our queuing network. MVA also has the advantage that approximations are known that extend the class of networks that can be solved well beyond the class of BCMP-type networks. MVA allows each heuristic search to be completed in a few seconds of CPU time. The computational complexity of each search is \( O(M \times K \times (K+N) \times D) \) where \( M \) is the number of components, \( K \) is the number of network devices, \( N \) is the number of communications links, and \( D \) is the number of tasks. It appears that substantial problems of this type could be solved using the methods we describe. We can compute the mean service times and queue sizes for different tasks at different devices and communication links.

### 5.3. Perspective Contract Checking

Let \( t_{s,i,j} \) be the mean service time for task \( i \) at device \( j \). Let \( t_{m}\), \( t_{k}\) be the mean service time for task \( i \) transferring data between \( j \) and \( k \) components at different devices. Let \( Q_{i,j} \) be the mean queue sizes for task \( i \) at device \( j \).

For each task \( i \), there exist an executing deadline \( T_d \), before which demands every instance must be finished. That means the life cycle time of every task instance from starting to ending must be less or at least equal than executing deadline. The temporal perspective contract of the system could be defined as:

\[
\sum_{m \in U} t_{s,i,j} + \sum_{j \in U} t_{m,i,k} \leq T_d, i = 1,2,...,D
\]

The right part of expression shows that the life cycle time of a task instance includes service time consumed at different devices and services time consumed at corresponding communication links among these devices. Our temporal perspective contract considers the network bandwidth consumption at the same time.

For each device \( j \), there exists a memory capacity limitation \( D_{M} \). The sum of memory consumption of the components at a device should not exceed the memory capacity limitation of this device. The memory perspective contract of the system could be defined as:

\[
\sum_{m \in U} \sum_{j \in U} (em_{m,j} / r_{m,j}) \leq D_{M}, j = 1,2,...,M
\]

From perspective contract checking, the developers know whether the solution can satisfy non-functional properties of the system in the scenario-modeling phase. If not, the developers can easily justify the solution to meet non-functional properties of the system because perspective contract checking provides not only simple answer like yes or no but also valuable information about the whole networked embedded system. From queuing network solution, the developers know the detail performance and resource consumption measures. They will easy find which device or which communication link is overloaded. They can expand the capacities of some overloaded devices or communication links. They also can move a component pre-allocated in a device to other device to ease the former device’s load or move two components pre-allocated in different devices to the same device to ease communication demands. Comparing with runtime checking, the justification in the scenario-modeling phase is more cost-effective and consumes less time.

### 6. Related Work

With the proliferation of enterprise component technologies, e.g., the CORBA Component Model (CCM) [5], Microsoft .NET [6], and Enterprise Java Beans (EJB) [7], large-scale distributed applications are increasingly being developed and deployed in a modular fashion. These technologies have many advantages including reusability of software and higher reliability since the components are written by domain experts.

There is also a lot of research done on the component-based solutions for networked embedded software, which builds upon these enterprise component technologies. Systems such as Real-time Object Oriented Modeling (ROOM) [8] [9], the Graphical Development Environment for TinyOS (GRATIS) [10], Component-Integrated ACE ORB (CIAO) [11], Virtual Component [12] appears to facilitate component-based design for networked
embedded software. However, none of these systems have adequate non-functional analysis capabilities.

The Composer Tool developed in the Software Engineering for Embedded Systems, using a Component Oriented Approach (SEESCOA) [13] project enables the construction of networked embedded software by means of components and contracts. An explicit construct (a contract) for annotating non-functional constraints has been defined. Contracts have a specification and a runtime meaning: they are used to annotate constraints at design time, and to monitor these at runtime. Although this runtime monitoring of contracts is valuable if one wants to detect non-functional failures at runtime, it does not prove the correctness of the application. Static contracts verification capacities are not provided.

Virginia Embedded Systems Toolkit (VEST) [14] provides an environment for constructing and analyzing component-based distributed real-time embedded systems. VEST helps developers select or create passive software components, compose them into a product, map the passive components onto active structures such as threads, map threads onto specific hardware, and perform dependency checks and non-functional analyses by aspects check to offer as many guarantees as possible along many dimensions including real-time performance and reliability. Now VEST currently focuses on the specific composition and analysis task, which does not support a top-to-bottom requirements specification and design methodology. To carry aspects check, some detail parameters of the components and platforms are necessary in the design phase.

Prediction-Enabled Component Technology (PECT) [15] is a development infrastructure that incorporates development tools and analysis techniques. PECT focuses on analysis; in principle any analysis could be incorporated. However, the framework does not include any theories on how to analyze different properties, just definitions of how analysis shall be applied in a so-called reasoning framework. To be able to analyze using PECT, proper analysis theories must be found and implemented and a suitable underlying component technology must be available.

7. Conclusion and Future Work

In this paper, we have described how networked embedded software is built by using component-based approach. The component based models allow one to subdivide the application into coherent parts, components. An explicit construct (a perspective contract) for annotating non-functional constraints has been defined. A queuing network based algorithm performs a static analysis on the perspective contracts before execution of the application.

The approach that was presented in this paper is based on some well-known principles: components, constraint, contracts, and static checking support. We have combined and extended these basic principles, to enable the construction of networked embedded software with support for the specification and static verification of non-functional constraints.

Our future work includes improving the tool to deal with additional constraints and optimize the analyzing algorithm. In addition to the non-functional constraints treated in this paper, there are other various types of non-functional constraints such as energy, cost, it would be challenging and interesting to consider multiple heterogeneous non-functional constraints, some times inter-dependent and conflicting as a whole. The queue based analyzing algorithm need to be improved to handle increasing complexity.

8. References


