A Formal Model for Component-Based Embedded Software Development

Changde Li, Xingshe Zhou, Yunwei Dong, Zhiwen Yu
Northwestern Polytechnical University
Xi’an Shannxi, China
lichangde@mail.nwpu.edu.cn, {Zhouxs, Yunweidong, Zhiwenyu}@nwpu.edu.cn

Abstract—This paper presents a formal model for specification, verification, and composition of component-based embedded software. We describe how components are specified from the syntactical view, functional view, QoS view and synchronization view. The refinement rules for functionality, QoS, and synchronous behavior are defined for the verification purpose. And a lightweight composition method is provided for the purpose of composition.

Keywords-component; contract; connector; embedded software; non-functional properties

I. INTRODUCTION

Embedded software is becoming more and more complex, costly and error-prone. All these trends pose an urgent need for advanced embedded software development techniques. However, the state-of-the-art in software development for embedded systems is far behind other application areas. Developing from scratch is common practice and Component-Based Software Development (CBSD) is usually a foreign word [1]. Potential benefits of component-based development such as reduction in development time and cost, improvement of quality, and specialization of expertise are as attractive in the domain of embedded system as they are in other areas of the software industry [2]. However, direct use of general-purpose component-based technologies in the field of embedded system is usually not feasible due to several key challenges. First, embedded applications are usually used in real-time and safety critical area. Therefore the non-functional properties (NFPs), such as reliability, real-time and memory constrains, have to be modeled explicitly to enable automated compositional reasoning. When building new applications from existing components it is not only necessary to ensure that they behave as expected, but also that these NFPs are fulfilled. Second, due to small size and requirements for mobility, embedded systems often have limited resources, e.g., processing power, storage capacity and network bandwidth. The application of a heavy-weight middleware (e.g., LPC for COM, ORB for CORBA, and JVM for JavaBeans/EJB) is often disadvantageous because of the limited resources in embedded systems [3]. A lightweight composition method is required. Third, embedded systems are often used in safety or mission critical area which requires the system to be modeled formally. Once a formal model is established, substitution of component can be precisely defined in terms of refinement. Informal or semi-formal models are usually not enough in embedded software fields for their weakness in semantic precision.

Our project aims to facilitate component-based software development for embedded systems. We establish a formal model for component-based software development in embedded application fields which we called ESCM (Embedded Software Component Model). The ESCM framework consists of three essential elements: interface, component, and connector. In ESCM the NFPs contract is provided to specify the non-functional attributes. And combined with the signature contract and functionality contract, they give a formal description of the interface. Interaction contract and glue contract are given for component and connector respectively, which describe the dynamic behavior and composition protocol of components. They can be used as a lightweight method for component composition. All the contracts in ESCM support compositional reasoning and verification for their well founded semantics.

The rest of this paper is organized as follows. In Section II we summarize the related work of CBSD for embedded software. Section III presents the framework of ESCM through three essential elements: interface, component, and connector. In Section IV we give the description of the formal contracts and their refinement rules. A case study is presented in Section V. Finally we conclude and point out directions for future work in Section VI.

II. RELATED WORK

To make full use of CBSD in embedded system development, several approaches have been proposed in the recent years. AADL [4] supports the specification and analysis of interactions between architectural components especially the description of NFPs. However, AADL lacks formal and unambiguous semantics; therefore detecting bugs, incompleteness and inconsistencies become difficult tasks. Although Wright [5] uses a variation of CSP algebra to specify the behavior and coordination of components, it does not address how to describe the NFPs. PECOS [1] attempts to enable component-based technology for a certain class of embedded systems known as “field devices”. The main features of this model are the data-flow-oriented programming style and the explicit incorporation of non-functional requirements. However, PECOS merely supports the NFPs of memory consumption and real-time, does not provide any formal verification supports, and only supports data type interface. These shortages make it not widely used. Although the Koala [6] component model is used for
embedded software in consumer electronic devices, which allows late binding of reusable components with no additional overhead, it does not take the NFPs into account and lacks a formal model.

III. THE FRAMEWORK OF ESCM

In order to apply component-based software development to embedded systems, we must precisely specify how components are structured and composed, which properties of components are important to capture and reason about, and how a component interacts with other components [1]. Here we present the framework of ESCM as shown in Fig.1.

We introduce our model through three essential notations: interface, component, and connector. Each element is contractually specified through one or more contracts and some additional information. Concretely, signature contract, functionality contract, and NFPS contract are responsible for interface, and arrow is used to identify the type of interface; interaction contract is responsible for components; and glue contract is responsible for connector.

A. Interface

Being a composite part of a larger system, a crucial feature of a component is its interface, as it describes the component’s interaction with the rest of the system in an abstract manner [7]. It is the interface that determines the external behavior and features of the component and allows the component to be used as a black box. We define an interface as a specification of exposed services through which components interact and can be used in constructing and maintaining embedded software.

**Definition1** (Interface): An Interface is a 4-tuple: \( I \equiv (Type, Signature, Functionality, NFPS) \), where,
- **Type**: is the role when the interface interacts with the component’s environment. A “provided interface” exposes a component’s functionality for usage by other components while a “required interface” specifies the need of functionality of other components.
- **Signature**: is the syntactic level specification of the interface describing how to use it. We give a signature contract supporting the syntactic specification.
- **Functionality**: is a semantic level specification of the interface’s functionality. It provides information about the effect of the interface. We present a formal description through functionality contract.
- **NFPS**: Many component models focus only on the functional aspects of a component. For embedded software, the non-functional constraints cannot be omitted. Modeling these non-functional constraints explicitly enables one to safely reuse components in a design, ensuring the non-functional constraints could be met. We model the NFPS specification through a NFPS contract.

B. Components

Component-based software development aims to build and maintain software systems by using existing software components. It is a common sense that components are required to be reusable and can interact with each other in a system architecture. For this purpose, an embedded software component is a reusable execution unit with contractually specified interfaces and coherent dynamic behaviors which are all formally specified.

**Definition2** (Component): A component is a 2-tuple: \( C \equiv (I, DB) \), where,
- **I**: represents a set of interfaces including provided interfaces and required interfaces. As interfaces are the only points of component interaction, a component must offer at least one provided interface and optional required interface. The definition of interface is given in Section III.A.
- **DB**: describes the dynamic behavior of component. It defines the protocols how the interfaces of the component interact with each other. We use an interaction contract to give a formal specification of the component’s dynamic behavior.

C. Connector

Generally speaking, connector mediates the communication and coordination activities among components. It provides the “glue” for architectural design. Connector is a synchronous constrain of components when building an application. We define connector as an explicit first class semantic entity specified in glue contract.

IV. CONTRACT SPECIFICATION

We establish our model based on the use of interface, component, and connector in combination with contracts. Five types of contracts are distinguished at four levels in ESCM, as depicted in Fig.2.

1. Signature contract is associated with interface. It presents syntactic descriptions of parameters for the interface operation.
2. Functionality contract is associated with interface and specifies services of components’ interface at functional level.
3. NFPS contract is associated with interface and specifies non-functional attributes at QoS level.
4. Interaction contract is associated with component and gives the behavior protocols of the interfaces the component included.
Glue contract is associated with connector. It can be seen as a light-weight middleware providing protocols for composing components into a system.

A. Signature Contract

We define a signature contract in the form of \( op \langle parameters \rangle \). This definition declares the interface operation name \( op \) and the list of its \( parameters \). Each parameter declaration is of the form \( x:T \), giving the name and type of the parameter. For the provided interface the \( parameters \) are the output variables of \( op \) but for the required interface the \( parameters \) are the input variables of \( op \).

B. Functionality Contract

The functionality of each operation \( op \) in the interface is a design that is specified as a pair of pre and post-condition of the form:

\[
P(x,FDec) \vdash Q(x',x,FDec,FDec')
\]

in Hoare and He’s Unifying Theories of Programming (UTP) [8], where non-primed and primed variables represent the values of the variables in the pre and post states respectively and \( FDec \) is the set of variables given in the field declarations. Precondition \( P \) characterizes the initial states of the interface and post-condition \( Q \) relates the initial states of the interface to its final states. If the precondition \( P(x,FDec) \) is true, the pair will be abbreviated as \( \vdash Q(x,x',FDec,FDec') \). It is proven that designs are closed under all imperative programming constructors such as assignment, sequential composition, conditional choice, recursion, and so on [8]. The syntax of the functional specification is as follows:

**Functionality**

1. **Declaration**
   - \( variable-name : data-type \)
   - \( // field variables’ declaration (FDec) \)
2. **Predicate**
   - \( Fun(op \langle parameter \rangle) = P \langle parameter, FDec \rangle \)
   - \( \vdash Q \langle parameter, FDec, parameter', FDec' \rangle \)

End Functionality

We define the alphabet \( \alpha \) as the union of the input variables \( ina \) and the output variables \( outa \). The refinement relation between designs is then defined to be logic implication.

**Rule 1** (design refinement): A design \( D_2 = (\alpha, P_2) \) is a refinement of design \( D_1 = (\alpha, P_1) \), denoted by \( D_1 \sqsubseteq D_2 \), if \( P_2 \) entails \( P_1 \), where \( x, x', .., z, z' \) are the variables in \( \alpha \).

Using the design refinement rule, the developer can determine whether a component can be substituted by another component at the functional level.

C. NFPs Contract

In the following, we describe how to model NFPs properties in the interface of a component, using the “Quality of Service Modeling Language” (QML) [9]. We select QML for two reasons. First, it can be used to define arbitrary quality dimensions. This is useful for embedded systems because the NFPs that the developer concerned may be diverse in different application filed. Second, QML defines conformance relation which can be used for the validation of substitutability of components at NFPs level.

QML consists of three main abstraction mechanisms for NFPs specification: contract type, contract, and profile. Contract type defines the dimensions that represent specific NFPs aspects, such as performance or reliability. Contract is an instance of a contract type and represents a particular NFPs specification. Finally, QML profiles associate contracts with the interface of component. Here we exemplify the use of QML by specifying reliability property.

\[
\text{type Reliability = contract} \{ \\
\text{Failures-number: decreasing numeric no/year; } \\
\text{TTR: decreasing numeric sec; } \\
\text{availability: increasing numeric; } \\
\};
\]

\[
\text{System-Reliability = Reliability contract} \{ \\
\text{Failures-number < 12 no/year; } \\
\text{TTR < 500; } \\
\text{Availability > 0.85; } \\
\};
\]

\[
\text{My-Interface-profile = profile} \{ \\
\text{require System-Reliability; } \\
\};
\]

In the reliability description above, the contract type defines three dimensions. The first one represents the number of failures per year. Time-to-repair (TTR) represents the time it takes to repair a service that has failed. Finally, availability represents the probability that a service is available.

QML defines a conformance relation on profiles and contracts. Profile conformance is defined in terms of contract conformance. A stronger specification conforms to a weaker specification. Essentially, a profile \( P \) conforms to another profile \( Q \) if the contracts in \( P \) associated with an entity \( e \) conform to the contracts associated with \( e \) in the profile \( Q \). For more details on conformance please refer to [9].

D. Interaction and Glue Contract

Interaction contract and glue contract are all behavior protocols. We distinguish them because the interaction contract describes the interaction protocol of a component while the glue contract describes the composition protocol of
a system. The interaction contract together with the glue contract forms a lightweight composition technique for embedded component software.

Hoare’s Communicating Sequential Processes (CSP) [10] is a well-founded formalism of progress algebra. We favor the use of CSP in the specification of interaction and glue contracts for two reasons [5]. First, it provides an ideal semantic basis to characterize the dynamic behavior of inter-component communication, to specify which components are responsible for making decisions during interaction and to detect mismatched assumptions that could cause a component to get “stuck” midway through its interaction with another component. Second, CSP’s parallel composition operator provides a simple but powerful form of composition.

In the language of CSP, a process describes an entity that can engage communication events. Events may be primitive, or they can have associated data (as e?x and e!x, representing input and output of data respectively). Two additional events r and \n are used to describe internal actions of a process or to model the termination of a process, respectively. The set of events with which a process P can communicate is termed aP. e→P denotes a process that engages in event e and then becomes process P. P □ Q represents a process that behaves like P or Q where the choice is made by the environment while P ⊓ Q means a process that behaves like P or Q, where the choice is made by the process itself. Processes can be composed using the || operator. Further operators of CSP are presented in [10].

The semantics of a CSP process P can be determined by the failure divergence model which is defined as a 3-tuples \((\alpha(P), F(P), D(P))\). Where, \(\alpha(P)\) denotes the alphabet of process P; F(P) represents the sets of event sequences rejected by P and D(P) is the set of event sequences resulting in an infinite sequence of internal events. Based on the failure divergence model, it can be checked if a process Q is a correct refinement of a process P.

**Rule2** (process refinement): P and Q are two processes. We say that P is refined by Q, denoted by \(P \sqsubseteq Q\), if

1. P and Q have the same communicating events set, i.e. \(\alpha(P) = \alpha(Q)\).
2. Q is not easier to diverge than P, i.e. \(D(Q) \subseteq D(P)\).
3. Q is not easier to deadlock than P, i.e. \(F(Q) \subseteq F(P)\).

In the specification of interaction and glue contract, interface of component is transformed into communication event in CSP. For the required interface \(op(parameters)\) we translate it into a CSP communication event of the form \(op?(parameters)\) while for the provided interface \(op(parameters)\) of the form \(op!(parameters)\). For the interface does not have any parameters we merely use a primitive event \(op\) to represent it. Then the interaction contract and glue contract are all modeled as a CSP process which is an entity that can engage communication events. The examples of interaction and glue contract are given in our case study in Section V.

## V. CASE STUDY

We use a fire alarm system to demonstrate the usage of the ESCM framework. The fire alarm system includes two components: the sensor component for detecting the temperature and the signal component for giving alarm signal when fire happens. As illustrated in Fig.3, the sensor component includes two interfaces: acquire interface for detecting the environment temperature and send interface for sending the data of temperature to alarm component; the alarm component also has two interfaces: get interface for receiving the data of temperature from sensor component and signal interface for giving alarm signal when it determines fire happens. ESCM is realized through the extended markup language (XML). XML language has been used because it provides supports for extensibility, flexibility, and serves as a means to support the specification syntax by using schemas [11].

![Fire-alarm System](image)

Figure 3. Fire-alarm System.
In the fire alarm system described above, we only give a functionality specification of the acquire interface and omit the QoS specification which is given in Section III.C for the length reason. Based on the behavior specification of component Sensor, Alarm and connector Glue, we can define a lightweight composition of the fire alarm system through the process composition operator " || ".

System Architecture of Fire Alarm System

We can define a lightweight composition of the fire alarm system through the process composition operator " || ":

Fire-alarm = sensor:Sensor || alarm:Alarm || Glue

That gives semantics to the combination of components and connector.

The steps of component-based development for the fire alarm system are as follows:

1. Specification: give the specification of the firm alarm system, including the sensor component, the alarm component and the connector. The XML profile is responsible for this purpose.
2. Validation: find out proper components from component repository that match the sensor component and alarm component. If there are not existing usable components then develop new ones according to the specification. Refinement rules for functionality, NFPs and dynamic behavior are used to validate the substitutability of components.
3. Composition: compose the sensor component and alarm component into a fire alarm system use the connector. The lightweight composition method is used to complete this mission.
4. Prediction and analysis: analyze properties of the fire alarm component especially the NFPs. This requires transforming ESCM model into an executable property analysis model.

VI. CONCLUSIONS AND FUTURE WORK

In this paper, we presented an approach for building component-based embedded software. We use five contracts to give a formal specification of ESCM from four separated levels view, including signature contract at syntactic level, functionality contract at functional level, NFPs contract at QoS level, and interaction and glue contracts at synchronization level. This strict specification is required if one wants to safely reuse component. When building a system, developers can verify the usability and correctness of a component through the design refinement rules for functionality in UTP, conformance rules for NFPs in QML, and the process refinement rules for synchronous behavior in CSP. Furthermore, ESCM presents a lightweight composition method of components based on the synchronization behavior using CSP’s parallel composition.

Our ongoing research includes handling the model transformation for analysis and prediction of NFPs of the application systems, e.g., translating it into a Markov Chain model for the verification of reliability; giving a formal architecture style of product family for the reusability of system architecture; and developing tools to support the establishment of the model.

ACKNOWLEDGMENT

This paper is supported by the National Natural Science Foundation of China under Grant No.60736017 and the National High-Tech Research and Development Plan of China under Grant No. 2007AA010304.

REFERENCES