International Journal of Control and Automation
Vol. 8, No. 6 (2015), pp. 293-306
http://dx.doi.org/10.14257/ijca.2015.8.6.29

ECM: An Formal Embedded Component Model for Embedded System

Ruiqin Sun[1], Guanzhong Yang[2], Dafang Zhang[2] and Bowen Yang[3]

[1]*College of Computer Science and Electronic Engineering, Hunan University, Changsha, 410082, China, ruiqin820@hotmail.com*
[2]*College of Computer Science and Electronic Engineering, Hunan University, Changsha, 410082, China*
[3]*School of Information Science and Engineering, Central South University, Changsha, 410082, China*

Abstract

At present, embedded system development establishes embedded component model (ECM) with informal and unstructured methods, and the dependencies among components are implicit and lack rigorous semantic. This paper presents a formal model for specification, verification, and composition of embedded system. In this paper, we provide a rigorous and compact description of the modeling elements of ECM and define component combination operations based on formal semantics which is feasible and effective both theoretically and practically. The corresponding techniques such as component property, connection constraints and ECM hierarchy are also discussed in this paper. Based on this model, DTSED component production toolkit is developed as the support platform for the ECM. Finally, a case study is introduced to explain how the above ECM can be used.

Keywords: *Embedded System, Embedded Component Model, Component Combination, Formal Semantic*

1. Introduction

The increased complexity of embedded systems leads to increasing demands with respect to requirements engineering, high-level design, early error detection, productivity, integration, verification and maintenance. This calls for models, and toolkit which permit a controlled and structured working procedure during the complete life cycle of the system. Component-based Software Engineering (CBSE) is a promising approach to improve quality, achieve shorter time to market and to manage the increasing complexity of software [1, 2]. When applying component-based software engineering (CBSE) methodology on the development of embedded systems, an important factor is reusability of components [3-5]. Currently, people have made research results component model in various aspects, including PBO (Port Based Object)[6], PECOS (Pervasive Component Systems)[7], Koala[8], Rubus[1] and so on. However, most of these studies used unstructured manner for analysis and modeling components from the perspective of software components[9], making them difficult to accurately express the component model for embedded systems[10, 11]. Therefore, research on semantics reasoning for the component model, which can establish a foundation to capture the essential properties and mimics the operation of the component to analyze, validate, and simulate component-based embedded systems that is both necessary and beneficial. While formal semantics of components can help rigorous reasoning and validation of target system, it can also ensure correctness of system composition [12].

Due to their nonlinear, multivariate and strong coupling (between the radial suspension forces) features, independent and accurate control for the suspension forces must be realized to enable a basic, stable working condition of a bearing less motor. The sliding mode variable structure control method possesses unique advantages in solving multivariable and nonlinear-coupled problems; hence, this paper applies it to the control of bearing less permanent magnet synchronous motors. Further, nonlinear differential geometry was employed to carry out the decoupling control and linearization of the radial suspension forces. Thus, the original multivariable system was transformed into a non-coupled, pseudo-linear subsystem with two independent radial positions, and then a sliding mode controller with fractional order was designed for the decoupled pseudo-linear subsystem using a neural network. From the simulation results, the feasibility of this kind of decoupling control method was validated, and the control quality and robustness of the system was verified. This paper presents a formal embedded component model to facilitate embedded systems development, which we called ECM (Embedded Component Model). On the basis of the existing component model, this paper shows the conceptions of embedded component model (ECM) elements and presents the corresponding formal semantics. In addition, connectors, component property, connection constraints and other issues are also discussed. Component combination operations are defined as arithmetic between components that ensure the components in different levels of abstraction and a clear and strict definition of their relationship. Then the corresponding techniques such as component property, connection constraints and ECM hierarchy are also discussed. Next, a formal algebraic model of ECM is proposed which can provide theoretical support for the design and implementation of embedded systems.

The ultimate goal of ECM is to guide the development of embedded systems, which requires a good support platform that can be used to achieve component production, reuse and management. This paper develops DTSED component production toolkit as a support platform for the embedded component model that supports the implementation, validation, assembly, execution and management of components, which can help understand, track and reuse the ECM.

The rest of this paper is organized as follows: Section 2 introduces the formal notation and the actual formalization of the constituent elements of ECM and defines some component operations by using domain theory. Section 3 shows the hierarchy of ECM. The supported platform for ECM and a case study of ECM are carried out in Section 4. Finally, Section 5 provides concluding remarks and directions to future work.

2. The Embedded Component Model

The ECM is the abstract description of the essential characteristics of components and component relationships, which concerns the interfaces of component and packages the implementation details of component. ECM is the basis and guarantee of component composition, and provides a consistency description of component that can be accepted by different components producers and users. ECM provides a rigorous and compact description of the modeling elements and de-fines component combination operations as arithmetic between components. ECM proposed in this paper includes component, connector and interface, component property and other elements. The following will describe details of various component elements.

2.1. Components

In ECM, a component is a software entity with relatively independent functions, contractually specified interfaces and explicit context dependencies that can be independently deployed and composed without modification according to a composition standard. A component definition describes the provided and required interfaces, the internal structure and the properties of the component, and is the unit of reuse. Formal semantics of components are foundations for rigorous analyzing and reasoning about the composition process and its correctness.

In ECM, a component is a software entity with relatively independent functions, contractually specified interfaces and explicit context dependencies that can be independently deployed and composed without modification according to a composition standard. A component definition describes the provided and required interfaces, the internal structure and the properties of the component, and is the unit of reuse. Formal semantics of components are foundations for rigorous analyzing and reasoning about the composition process and its correctness.

Define 1(Component, abbreviation C).A component X is defined as a 4-tuple $X = \langle I_X^P, I_X^R, InS_X, \imath$,where,

- : a set of provided interfaces that constitute the component X.
- : a set of required interfaces that constitute the component X. $I_X^P \cap I_X^Q$. Define all external interfaces set as $I_X = I_X^P$.
- : the internal structure of component X, the internal structure is not visible outside. is defined as a 2-tuple $InS_X = \langle C_X,$,where is the set of subcomponents that constitute the component X, and is the set of connectors that constitute the component X.
- :a set of property of the component X, called property bundles of X(e.g., component id, component name).

As in Koala, an interface provides access to the functionality of a component, or it specifies the use of (external) functionality of a component. For a component, the provide interface is essential and the require interface is not, because some components do not need any service provided by any external components, therefore I_X^R (the set of require interfaces) of those components is empty.

As already mentioned earlier, ECM components explicitly declare the require interfaces and provide interfaces. An interface is a small set of semantically related elements referred to as services, which can be functions, parameters or constants, collectively. An interface definition describes the semantics of these elements, and is the unit of specification.

Define 2 (Interface). An interface $I_{X_i}(\forall I_{X_i} \in$ is defined as a 3-tuple $I_{X_i} = \langle IName, IType, IServic$, where ,

- IN: the name of a interface.
- $I\Gamma$: the interface type ,which is provide or require.
- $ISer\imath$: a set of semantically related services of the interface, which can be functions, parameters or constants.

In ECM, a connector defines the connectivity between two or more components through the interfaces of these components. Connectors are mainly used to implement static binding, compile-binding and run-time binding. Connector type itself is scalable, according to the different functionality can be divided into several different types, this paper mainly presents three basic connectors: direct connector, data process connector, switch connector.

A direct connector connects a component to other components through a provide interface and a require interface directly without any other process. A data process connector connects a component to other components through a provide interface and a require interface, which provides simple data process such as field-format conversion, simple data operation, *etc*.

A switch connector connects a list of components to a specified component in order to realize the interaction between multiple components. Logically, switch connector can be seen as a switch-case multi-branch open expression. One side of the switch connector has an interface of one component, called O-side. The other side has multiple alternative interfaces of multiple components, called M-side, where switch connector select one of a plurality of interfaces to interact.

Direct connectors and data process connectors are used to implement static binding and switch connectors are used to implement compile-time and run-time binding.

Define 3 (Connector). An connector $Cn_{x_i}(\forall Cn_{x_i} \in C$ is defined as a 5-tuple $Cn_{x_i} = \langle CnId, CnType, ProIn, ReqIn, P$, where

- : the identification of a connector.
- CnT: the connector type ,which can be Direct, Data_processor Switch.
- P:a set of the provide interfaces of those components connected by this connector,

and $ProIn$.

- R:a set of the require interfaces of those components connected by this connector,

and $ReqIn \neq \phi, ProIn \cap ReqIn$.

- : a processing element of the connector. For example, data process connector depends on PE to accomplish data conversion and other processing work.

In ECM, basic authentication model validation can be detected by relying on the following rules.

Rule 1 (Interface quantity of connector). Let A is a connector. If the type of A is Direct or Data process, A will have one and only one provide interface and one and only one require interface. Otherwise, if the type of A is Switch, then it will have one provide interface and at least two require interfaces or at least two provide interface and one require interface. This rule is formally defined as:

- $(a.CnType = Direct) \vee (a.CnType = Data\_Process) \Leftrightarrow |a.P$= $|a.R$=1.

  $a.CnType = Switch \Leftrightarrow ((|a.ProIn| = 1) \wedge (|a.ReqIn| \geq 2) \vee (|a.ProIn| \geq$
- $2) \wedge$ $(|a.ReqIn| = 1))$

Rule 2 (PE Existence of Connector). Let a be a connector. If type of a is Direct, then its PE must not exist. Otherwise, if type of a isData_process, then its PE must exist. This rule is formally defined as:

- $a.CnType = Direct \Rightarrow a.PE.$
- $a.CnType = Data\_process \Rightarrow a.PE.$

Rule 3 (Interface connection protocol). A connection cannot be established between a provide interface and a require interface from a same component. Let a be a connector of ECM, Let A be a component of ECM, $\forall i \in I_A^P$, $\forall j \in$. This rule is formally defined as:

- $\forall i \forall j (i \in I_A^P, j \in I_A^R) \Rightarrow \nexists a\{a | (i \in a.ProIn) \wedge (j \in a.Re$

In ECM, There are three categories of component property: metadata property, trustiness property, and system property. Metadata property shows the most basic attribute of component, such as component Id and component name or sample component description information. Trustiness property reflects some aspect of the credibility of the components, including the function property and non-function property: Function properties describe the services provided by the component, or the reactions made by the component in a particular input or in a particular scene.

Non-functional properties are some of the constraints on the functional properties of the component, the whole nature of component or component development process, such as performance, reusability, maintainability, security and reliability, *etc*.

System properties are the hardware information associated with the component. In whole ECM, using technical drawings (*i.e.*, certain established standards and principles drawn diagram of hardware components) to describe the system properties, what will help facilitate developers to learn more about the physical structure of the ECM.

Define 4 (Component Property). An property bundles of a component X is defined as a 3-tuple $P_X = (\langle Id, P_X^M\rangle, \langle P_X^F, P_X^{Non}\rangle, I$, where,

- : the identification of X.
- $\langle Id$: a set of the metadata property of X.
- $\langle P_X^F, P$: a set of the trustiness property of X, where is a set of the function

property of X, and is a set of the non-function property of X.

: a set of the system property of X.

If a component X contains n subcomponents (n>=1): X1,X2, . . . Xn, then its properties will depend to some extent on its internal sub-components' properties. Here to show such a relationship represented by an n-expression, namely:

$$P_X = F_X(P_{X_1}, P_{X_2}, \dots\dots, P_{X_n})\ (X_i \in C_X, i \in [1,n], n = |C_X|,\ C_X \neq \emptyset)$$

The above expression is a generalization. The relationship between components may be a simple sum, may be exponential changes and even may be not exit. For a particular property, expression needs specific analysis based on specific conditions.

2.2. Component Operations

A problem in component-based development is often encountered that the existing component cannot meet the requirements of users, which often influences user efficiently component reuse. This is mainly because a single component cannot meet all the requirements of users and it need be solved by the component combination. By understanding the component combination of an operation that is a new idea and extending the calculus in algebraic process, a variety of component operations are defined, such as components add, delete, and connect operation by using domain theory, which can achieve effective reuse components. Based on formal semantics, component combination which is proposed in this paper is feasible and effective both theoretically and practically.

Define 5(Dot operation). Dot operation extracts parts of component. For example, let *A* is a component in the domain Dom(U) and*irA* is an interface of A, then *irA*can be expressed as *A.irA* .

Define 6 (Equality relation of components). For any two components A, B in the domain Dom(U) are equal if and only if I_A^P =and I_A^R =,denoted by A ⇔ B, where " ⇔ " a meta logical symbol representing the logical equivalence.

Define 7(Direct connection operation). Let A, B be the components in the domain Dom(U) $ipA \in I_A^P,\ irB \in$,. The process that connecting interface i, i through a direct connector, called a direct connection operation between the A and B, denoted by $A^{d(ipA,irB}_{\overrightarrow{\otimes}}$, where, d is the shorthand for direct, and brackets contains two parameters are a provide interface and a require interface of the direct connector . This process generates a direct connector, $Cn_{A^{d(ipA,irB)}_{\overrightarrow{\otimes}}B} = \langle CnId, Direct, \{ipA\}, \{irl$

When $B \notin$, $A^{d(ipA,irB}_{\overrightarrow{\otimes}}$ can be seen as a whole .Then $A^{d(ipA,irB}_{\overrightarrow{\otimes}}$ is still a component, denoted by AΔB, and satisfies the following properties:

$$\begin{cases} I_{A\Delta B}^P = (I_A^P \cup I_B^P) - \{ipA\} \\ I_{A\Delta B}^R = (I_A^R \cup I_B^R) - \{irB\} \\ InS_{A\Delta B} = \langle C_{A\Delta B}, Cn_{A\Delta B}\rangle \\ C_{A\Delta B} = \{A, B\} \\ Cn_{A\Delta B} = \left\{ Cn_{A^{d(ipA,irB)}_{\overrightarrow{\otimes}}B} \right\} \end{cases}$$

Define 8 (Date Process connection operation). Let A, B be the components in the domain Dom(U), $ipA \in I_A^P,\ irB \in$. The process that connecting interface i,

i through a date process connector, called a direct connection operation between the A and B, denoted by $A\overset{p(ipA,irE}{\vec{\otimes}}$, where, p is the shorthand for date_process, and brackets contains two parameters are a provide interface and a require interface of the direct connector.

This process generates a direct connector, $Cn_{A\overset{p(ipA,irB)}{\vec{\otimes}}B} = \langle CnId, Direct, \{ipA\}, \{ir l.$

When B, $A\overset{p(ipA,i}{\vec{\otimes}}$ can be seen as a whole .Then $A\overset{p(ipA,i}{\vec{\otimes}}$ is still a component, denoted by , and satisfies the following properties:

$$\begin{cases} I^P_{A\nabla B} = (I^P_A \cup I^P_B) - \{ipA\} \\ I^R_{A\nabla B} = (I^R_A \cup I^R_B) - \{irB\} \\ InS_{A\nabla B} = \langle C_{A\nabla B}, Cn_{A\nabla B} \rangle \\ C_{A\nabla B} = \{A, B\} \\ Cn_{A\nabla B} = \left\{ Cn_{A\overset{p(ipA,irB)}{\vec{\otimes}}B} \right\} \end{cases}$$

Define 9(Switch connection operation). Let A, $B_1, \ldots\ldots, B_n$ be the components in the domain Dom(U), $ipA \in I^P_A, irB_i \in I^R_{B_i} (1 \le i \le$. The process that connecting interface i, $irB_1 \ldots\ldots irB_n$ through a switch connector, called a switch connection operation between the A and $B_1, \ldots\ldots, B_n$, denoted by $A\overset{s(ipA,irB_1 \ldots irB_n)}{\vec{\otimes}}(B_1 \cdots l$, where, s is the shorthand for switch, and brackets contains n +1 parameters are the interfaces of the switch connector . This process generates a switch connector, $Cn_{A\overset{s(a,b_1 \ldots b_n)}{\vec{\otimes}}(B_1 \cdots B_n)} = \langle CnId, Switch, \{ipA\}, \bigcup_{i=1}^{n}\{irB_i\}, F.$

When $B_i \notin C_A (i = [$, $A\overset{s(ipA,irB_1 \ldots irB_n)}{\vec{\otimes}}(B_1 \cdot\cdot$ can be seen as a whole. Then $A\overset{s(ipA,irB_1 \ldots irB_n)}{\vec{\otimes}}(B_1 \cdot\cdot$ is still a component, denoted by $A \bowtie (B_1 \cdot\cdot$, and satisfies the following properties:

$$\begin{cases} I^P_{A\bowtie(B_1 \cdots B_n)} = \left(I^P_A \cup \bigcup_{i=1}^{n} I^P_{B_i} \right) - \{ipA\} \\ I^R_{A\bowtie(B_1 \cdots B_n)} = \left(I^R_A \cup \bigcup_{i=1}^{n} I^R_{B_i} \right) - \bigcup_{i=1}^{n} \{irB_i\} \\ InS_{A\bowtie(B_1 \cdots B_n)} = \langle C_{A\bowtie(B_1 \cdots B_n)}, Cn_{A\bowtie(B_1 \cdots B_n)} \rangle \\ C_{A\bowtie(B_1 \cdots B_n)} = \{A\} \cup \bigcup_{i=1}^{n} \{B_i\} \\ Cn_{A\bowtie(B_1 \cdots B_n)} = \left\{ Cn_{A\overset{s(ipA,irB_1 \ldots irB_n)}{\vec{\otimes}}(B_1 \cdots B_n)} \right\} \end{cases}$$

Define 10(Add operation). Let A, B be the components in the domain Dom(U). Add operation is defined as the process adding B into A as a subcomponent, denoted as $A \oplus$ After executing $A \oplus$, component A can be expressed as $A \oplus B = \langle I^P_{A\oplus B}, I^R_{A\oplus B}, InS_{A\oplus B}, P_{A\oplus}$ and satisfies the following properties:

$$\begin{cases} I^P_{A\oplus B} = I^P_A \\ I^R_{A\oplus B} = I^R_A \\ InS_{A\oplus B} = \langle C_{A\oplus B}, Cn_{A\oplus B} \rangle \\ C_{A\oplus B} = C_A \cup \{B\} \\ Cn_{A\oplus B} = Cn_A \\ P_{A\oplus B} = F_A\left(P_B, P_{A_1}, P_{A_2}, \dots\dots, P_{A_n}\right) \\ \left(A_i \in C_{A\oplus B} - \{B\}, i \in [1,n], n = \left|C_{A\oplus B}\right| - 1\right) \end{cases}$$

After adding the component B into component A, the interfaces set of A remain unchanged, and the connectors set of A will not change unless direct operation, data process operation and switch operation (hereinafter collectively referred to as connect operations). But due to the addition of B, the property set of A may be changed.

(Delete operation). Let A, B be the components in the domain Dom(U) and $B \in$. Let Cbe a set of connectors which have the interface of component B, and $Cn'_A \subseteq C$.
Delete operation is defined as the process deleting B from A anddeleting C from Cat the same time, denoted as $A \ominus$. After executing $A \ominus$, component A can be expressed as $A \ominus B = \langle I^P_{A\ominus B}, I^R_{A\ominus B}, InS_{A\ominus B}, P_{A\ominus}$ and satisfies the following properties:

$$\begin{cases} I^P_{A\ominus B} = I^P_A \\ I^R_{A\ominus B} = I^R_A \\ InS_{A\ominus B} = \langle C_{A\ominus B}, Cn_{A\ominus B} \rangle \\ C_{A\ominus B} = C_A - \{B\} \\ Cn_{A\ominus B} = Cn_A - Cn'_A \\ P_{A\ominus B} = F_A\left(P_{A_1}, P_{A_2}, \dots\dots, P_{A_n}\right) \\ \left(A_i \in C_{A\ominus B}, i \in [1,n], n = \left|C_{A\ominus B}\right|\right) \end{cases}$$

After deleting the component B from component A, the interfaces set of A remain unchanged, and the connectors, which contains the interfaces of component B, are removed from the connector set of A. Similarly, due to the “disappearance” of B, the property set of A may be changed.

Add operation and delete operation show that after adding a component into another component or deleting a component from another component, only its external visible the set of interfaces unchanged, other parts are changed accordingly.

2.3. Hierarchy of ECM

One of the most important goals of ECM design is components with high cohesion components and loose coupling. Then using component operations to assemble more complex, and higher level of abstraction of new components and ultimately form a complete embedded system. This embedded component model consists of three levels of components: atomic component, compound component and system.

Define 11(Atomic component). Atomic components are the components that exist solely to present, rather than to contain other components, denoted by X_Atom. X_Atom is formally defined as:

$$\begin{cases} X\_Atom = \langle I^P_X, I^R_X, InS_X, P_X \rangle \\ InS_X = \langle C_X, Cn_X \rangle \\ C_X = \emptyset \\ P_{X\_Autom} = \left(\langle Id, {P_X}^M \rangle, \langle {P_X}^F, {P_X}^{Non} \rangle, {P_X}^S\right) \end{cases}$$

Atomic component can exist independently, or as a subcomponent included in other components. In ECM, there is a special case of atomic component which has one and only one provide interface, no any property nor internal structure, such atomic component does not depend on other components to achieve one or more functions, called atomic implementation component, denoted by X_ImpleAtom. X_ImpleAtom is formally defined as:

$$\bullet X\_ \quad ImpleAtom = \langle I_X^P, \emptyset, \emptyset ..$$

Define 12 (Compound component). A compound component is built by assembling the existing components which may be atomic components and other compound component, denoted by X_Comp. X_Comp is formally defined as:

$$\begin{cases} X\_Comp = \langle I_X^P, I_X^R, InS_X, P_X \rangle \\ InS_X = \langle C_X, Cn_X \rangle \\ C_X = \bigcup_{1 \le j \le m} C_j \, (m = |C_X|) \\ Cn_X = \bigcup_{1 \le k \le h} Cn_{X_k} \, (h = |Cn_X|) \\ P_X = \left(\langle Id, P_X{}^M \rangle, \langle P_X{}^F, P_X{}^{Non} \rangle, P_X{}^S \right) \\ P_X = F_X \left(P_{X_1}, P_{X_2}, \ldots\ldots, P_{X_n} \right) (X_i \in C_X, i \in [1, n], n = |C_X|, C_X \neq \emptyset) \end{cases}$$

The subcomponents of compound component are not visible from outside. Each contained component has a type name–for example, *PressureController*and an instance name–for example, *pressurecon*. Between compound component and its subcomponents is a containment relationship, which is defined by the mapping among their interfaces. All require interfaces of a compound component can interact with a provide interface of other components, and each provide interface of a compound component can interact with zero or more require interfaces of other components. Types of interactive interface must be matched. It is important to note that when the compound component interacts with its subcomponents, the interface type of the compound component must be used as the opposite type. For example, Let *A* is the component and *irA* is a provide interface of *A* (*irA.IType = provide*). When *A* interacts with its subcomponents through *irA*, *irA* must be used as a require interface. This type of invisible transformation can guarantee the correctness of the whole ECM interaction.

All the definition of components in ECM will be stored in a global component repository, where each component has a globally unique name, used in component descriptions, in order to achieve effective management of components. Our component repository is flat which contains both atomic and compound component. Moreover, both components and connectors can be nested to form hierarchies; a higher-level component can be composed of several mutually interconnected and cooperative subcomponents.

According to the requirements of an embedded system, by executing finite number component operations among atomic components and compound components, a complete and well-structured ECM for the embedded systems can be established. Here, introduces the system initialization problem in ECM. In order to avoid unnecessary and duplicate initialization, the concept of faceted-oriented is introduced to handle system initialization problem. Design an atomic implementation component to complete the initialization of the whole system, regarded as an initialization facet. In order to maintain the overall hierarchy of ECM and facilitate the user to understand, we make some adjustments in the ECM graphical representation that the initialization interface of a subcomponent is connected to the initialization interface of its compound component to represent their initialization.

A complete embedded system formed by a large number of components assembled together through component operations.

Define 13(System). Let U be a domain:

•**(Closure)** Component instance is an integral element of the embedded system.

• **(Extensibility)** If new components satisfy the certain conditions, their instances can be added to the system by component operations.

•**(Hierarchy)** The result of the components operations is still a component and component instances execute finite number component operations to establish the embedded system.

Embedded Systems is denoted by ES = ⟨C, O⟩, where C is the set of components in the domain Dom(U), O represents the set of component operations in the domain Dom(U) among component instances of the components in C, any operation expressed byop_i, such as$\mathrm{op}_i = A \bigoplus B$.

3. Platform and a Case Study

The ECM proposed in this paper have been designed and implemented in a component platform, Domain-oriented &Trusted Embedded Software Development (DTSED), which is able to design and reuse components, build multiple embedded systems families, produce many high-value assets such as component repository.

DTSED component production toolkit based on Eclipse GEF framework and platform adopted industry and academia widely to support the design, development and assembly of interfaces, connectors, and components, and provides a friendly graphical interface, code generation and editing (the compiler related to the implementation language, such as C compiler), components test, components storage management and components monitoring toolkit.

So far, DTSED component production toolkit platform has been successfully applied in Hunan Micome Medical Co., Ltd.'s ventilator product line. We use a noninvasive ventilator product (ST-30C) to demonstrate the usage of the ECM. Due to the entire ECM of Noninvasive Ventilator is too large, the noninvasive ventilator Continuous Positive Airway Pressure embedded component model (CPAPECM) fragments is chose to explain how the ECM can be used (Figure 1). CPAPECM is clinically mainly used in the treatment of sleep apnea syndrome (SAS) and related diseases. In this statement that we use an instance of components selected from component repository to construct CPAPECM. Each component can be instantiated as multiple instances, such as in Figure 1, create an instance *cpap* of the CPAP component selected from component repository, and then use this instance to construct the model.

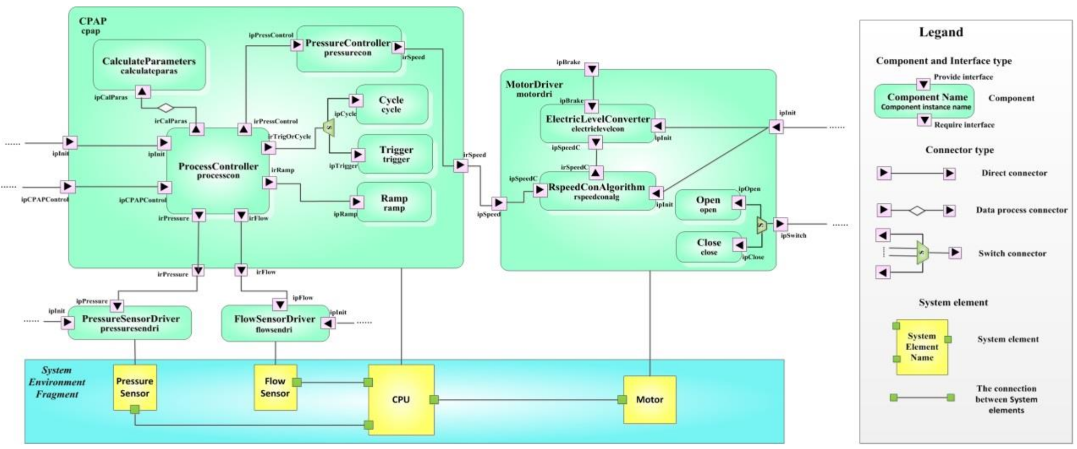


Figure 1. Noninvasive Ventilator Continuous Positive Airway Pressure Embedded Component Model (Cpapecm) Fragment

As shown in Figure 1, CPAPECM mainly contains four parts, they are an instance of CPAP component *cpap*, an instance of PressureSensorDriver component *pressuresendri*, an instance of FlowSensorDriver component *flowsendri* and an instance of MotorDriver component *motordri*. CPAP component is a compound component, primarily responsible for achieving noninvasive ventilator treatment in CPAP mode. PressureSensorDriver component is an atomic component,primarily responsible for converting an electrical signal intoa pressure output. FlowSensorDriver component is an atomiccomponent, primarily responsible for converting an electricalsignal into a flow output. MotorDriver component also isa compound component, mainly responsible for controllingthe speed according to the target pressure and the real-timepressure.

What's more, there is a technical drawings fragment of system environment, which shows the interaction within the system environment and the interaction between components and external system environment under CPAP mode. System environment impacts the system properties of components.

Here is how to use CPAP component as an example to describe the way component assembled. CPAP component is the core part of CPAPECM, it contains six subcomponents: an instance of CalculateParameters component *calculateparas*, used to parameter control; an instance of ProcessController component *processcon*, used to process control; an instance of PressureController component *pressurecon*, used to pressure control; an instance of Ramp component *ramp*, used to crawl control; an instance of Cycle component *cycle*, used to revocation control; an instance of Trigger component *trigger*, used to trigger control. The process of the construction of the CPAP component can use the following algebraic representation.

Known: CPAP component, CalculateParameter component (abbreviated CalParas), ProcessController component (abbreviated ProCon), PressureController component (abbreviated PreCon), Ramp component, Cycle component and Trigger component are components in the domain Dom(U). Formal representation of each component is as follows:

- $CPAP = \langle I^P_{CPAP}, I^R_{CPAP}, InS_{CPAP}, P_{CPAP} \rangle, I^P_{CPAP} = \{ipInit, ipCPAPCont$

$I^R_{CPAP} = \{irPressure, irFlow, irSpeed\}, InS_{CPAP} = \langle C_{CPAP}, Cn_{CPAP} \rangle,$

$P_{CPAP} = (\langle CPAP, P^M_{CPAP} \rangle, \langle P^F_{CPAP}, P^{Non}_{CPAP} \rangle, P^S_{CPAP}).$

- $CalParas = \langle I^P_{CalParas}, I^R_{CalParas}, InS_{CalParas}, P_{CalP}$

$I^P_{CalParas} = \{ipCalParam\}, I^R_{CalParas} = \emptyset, InS_{CalParas} = \langle \emptyset, \emptyset \rangle,$

$P_{CalParas} = (\langle CalculateParameter, P^M_{CalParas} \rangle, \langle P^F_{CalParas}, P^{Non}_{CalParas} \rangle, P^S_{CalP}.$

- $ProCon = \langle I^P_{ProCon}, I^R_{ProCon}, InS_{ProCon}, P_{ProCon} \rangle, I^P_{ProCon} = \{ipInit, ipCPAPCont$

$I^R_{ProCon} = \{irPressControl, irCalParam, irPressure, irFlow, irTrigOrCycle, irRamp\},$

$InS_{ProCon} = \langle,$

$P_{ProCon} = (\langle ProcessController, P^M_{ProCon} \rangle, \langle P^F_{ProCon}, P^{Non}_{ProCon} \rangle, P^S_{Pr}.$

- $PreCon = \langle I^P_{PreCon}, I^R_{PreCon}, InS_{PreCon}, P_{PreCon} \rangle, I^P_{PreCon} = \{ipPressCont$

$I^R_{PreCon} = \{irSpeed\}, InS_{PreCon} = \langle,$

$P_{PreCon} = (\langle PressureController, P^M_{PreCon} \rangle, \langle P^F_{PreCon}, P^{Non}_{PreCon} \rangle, P^S_{Pre}$

- $Ramp = \langle I^P_{Ramp}, I^R_{Ramp}, InS_{Ramp}, P_R, \quad I^P_{Ramp} = \{ipRa$

$I^R_{Ramp} = \emptyset, InS_{Ramp} = \langle, \quad P_{Ramp} = (\langle Ramp, P^M_{Ramp} \rangle, \langle P^F_{Ramp}, P^{Non}_{Ramp} \rangle, P^S_{R}.$

- $Cycle = \langle I^P_{Cycle}, I^R_{Cycle}, InS_{Cycle}, P_C, \quad I^P_{Cycle} = \{ipCy$

$I^R_{Cycle} = \emptyset, InS_{Cycle} = \langle \emptyset, \emptyset \rangle, P_{Cycle} = (\langle Cycle, P^M_{Cycle} \rangle, \langle P^F_{Cycle}, P^{Non}_{Cycle} \rangle, P^S_{C}.$

- $Trigger = \langle I^{P}_{Trigger}, I^{R}_{Trigger}, InS_{Trigger}, P_{Trigger} \rangle, I^{P}_{Trigger} = \{ipTrig$

$I^{R}_{Trigger} = \emptyset, InS_{Trigger} = \langle \emptyset, \emptyset \rangle,$

$P_{Trigger} = (\langle Trigger, P^{M}_{Trigger} \rangle, \langle P^{F}_{Trigger}, P^{Non}_{Trigger} \rangle, P^{S}_{Tri}$

According to the component operations as executed herein, the assembly process of CPAP component including six can be represented by such expressions:

1) $(CPAP \oplus CalParas) \| (CPAP \oplus ProCon) \| (CPAP \oplus PreCon) \| (CPAP \oplus Ran$

$(CPAP \oplus Cycle) \| (CPAP \oplus Trig.$

$\left(PreCon \overset{d(ipPressControl, irPressControl)}{\overrightarrow{\otimes}} ProCon \right) \| \left(Ramp \overset{d(ipRamp, irRamp)}{\overrightarrow{\otimes}} ProCo \right.$

$\left((Cycle, Trigger) \overset{s(ipCycle, ipTrigger, irTrigOrCycle)}{\overrightarrow{\otimes}} ProCon \right) \|$

$\left(CalParas \overset{p(ipCal\_Paras, irCal\_Paras)}{\overrightarrow{\otimes}} ProCon \right).$

2) $\left(CPAP \overset{d(irPressure, irPresure)}{\overrightarrow{\otimes}} ProCon \right) \| \left(CPAP \overset{d(irFlow, irFlow)}{\overrightarrow{\otimes}} ProCo \right.$

$\left(CPAP \overset{d(irSpeed, irSpeed)}{\overrightarrow{\otimes}} PreCon \right) \| \left(Pro\_Con \overset{d(ipInit, ipInit)}{\overrightarrow{\otimes}} CPAP \right) \|$

$\left(ProCon \overset{d(ipCPAPControl, ipCPAPControl)}{\overrightarrow{\otimes}} CPAP \right).$

Expression (1) describes the operations that add Calculate Parameter, Process Controller, Pressurecon, Ramp, Cycle, Trigger to CPAP component.

Expression (2) describes the component operations between the subcomponents of CPAP component, which produce two direct connectors, a data process connector and a switch connector, such as,

$$Cn_{Ramp \overset{d(ipRamp, irRamp)}{\overrightarrow{\otimes}} ProCon} = \langle RampCn, Direct, \{Ramp.ipRamp\}, \{ProCon.irRamp\} \rangle.$$

Expression (3) describes the component operations between CPAP component and its subcomponents (the interface type of the CPAP component must be used as the opposite type, as shown in the Section 3 compound component part), which produce four direct connectors, such as,

$$Cn_{CPAP \overset{d(rSpeed, rSpeed)}{\overrightarrow{\otimes}} PreCon} = \langle SpeedCn, Direct, \{CPAP.irSpeed\}, \{PreCon.irSpeed\} \rangle.$$

After a series of components operations above, here is the complete CPAP component. The above 9 connectors generated by expressions (2) and (3) composed Cn_{CPAP} . So,

$CPAP = \langle I^{P}_{CPAP}, I^{R}_{CPAP}, InS_{CPAP}, P_{CP}$, where,

- $I^{P}_{CPAP} = \{ipInit, ipCPAPCon$
- $I^{R}_{CPAP} = \{\{irPressure, irFlow, rSpe$
- $InS_{CPAP} = \langle C_{CPAP}, Cn_{C}$
- $C_{CPAP} = \{CalParas, ProCon, PreCon, Ramp, Cycle, Trig,$
- $\|Cn_{CPAP}\|,$
- $P_{CPAP} = (\langle CPAP, P_{CPAP}{}^{M} \rangle, \langle P_{CPAP}{}^{F}, P_{CPAP}{}^{Non} \rangle, P_{CP}$
- $P_{CPAP} = F_{CPAP}(P_{CalParas}, P_{ProCon}, P_{PreCon}, P_{Ramp}, P_{Cycle}, P_{Tri}$

Under CPAP model, the most important characteristic of CPAP component is the constant pressure output, and the value of this pressure changes in a certain range, called *p_pressurerange* ($p\_pressurerange \in P_{CPAP}{}^{\Lambda}$). It is determined by *p_pressurerange*', the pressure range of its subcomponent, Calculate Parameter($p\_pressurerange' \in P_{Cal\_Paras}{}^{\Lambda}$).In this case, $p\_pressurerange'$ ranges from $4cmH_2O$ to $25cmH_2O$ (*p_pressurerange'*[$4cmH_2O$, $25cmH_2O$]),so the *p_pressurerange*range of CPAP component also range from $4cmH_2O$ to $25cmH_2O$ (*p_pressurerange* [$4cmH_2O$, $25cmH_2O$]).

Through expression (1) ~ (3), we can clearly get CPAP component assembly process. When using DTSED component production toolkit to design ECM, the tool automatically executes the model validation according to Rule 1 to 3, to ensure the correctness and reasonableness of the ECM. On this platform, we can complete a series of work such as component design, component testing, and components storage, thus forming a large number of valuable reusable assets.

Hunan Micome Medical Co., Ltd. initiated DTSED Component Production Toolkit to install a software product line of Ventilator, which formed a component repository much 10^3 components to achieve the effective management and reuse of a large components, guided the development of the production of a variety of ventilator. The ECM approach yields a reuse rate of about 50% for new products. About 40% of the application components are based on slightly modified core assets and only 20% require writing new code. The reuse of the core components leads to significant business advantages. Compared with the development of earlier products, the development of new firmware takes only 50% of the staff resources. In spite of the reduction of staff, the development takes only 38% of the time. This productivity improvement goes hand in hand with a qualitative advancement. The ECM approach leads to 66% fewer defects compared with earlier products.

5. Conclusion

This paper presents a clear semantics embedded component model (ECM) intended to be used in embedded systems, including formal definition of component, interface, connector, component property, component combination and model hierarchy, which can provide theoretical support for embedded system design and development. There are three important points of the above study: use scalable connectors to achieve components communication, include hardware information into the scope of component property and explain component combinations as the implement of the component operations. The ECM presented in this paper not only enhances the ability to describe component model, but also improves the corresponding component management and application to a higher level of abstraction and reuse. Therefore, the ECM provides a consistent basis for the development of components-based embedded systems. Finally, DTSED component production toolkit is presented as component modeling support platform that follows the ECM and implements the execution environment for the components.

Interesting works still remains to be done. In the future, we will focus on constructing traceability between ECM and feature model for software product line engineering. Applying this ECM to dynamic software product line also is the important content of the next step of work.

Acknowledgements

This work is supported by the National Basic Research Program of China (973) (No. 2012CB315805), the National Science Foundation of China (No. 61173167), and the Special Foundation for the Industry-Academia-Research Key Science and Technology Program of Hunan Province, China (No. 2010XK6024).

References

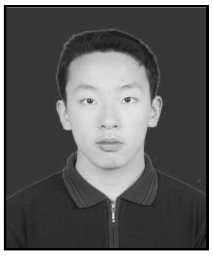
[1] D. Isovic and C. Norstrom, "Components in Real-Time System", Proceedings of 8th International Conference on Real Time Computing Systems and Applications, **(2002)**.
[2] A. K. Pandey and C. P. Agrawal, "Analytical Network Process Based Model to Estimate the Quality of Software Components", Proceedings of International Conference on Issues and Challenges in Intelligent Computing Techniques, **(2014)**, pp. 678-682.
[3] G. Kumar and P. K. Bhatia, "Comparative Analysis of Software Engineering Models From Traditional to Modern Methodologies", Proceedings of 2014 Fourth International Conference on Advanced Computing and Communication Technologies, **(2014)**, pp. 189-196.
[4] B. Jalender, D. A. Govardhan and D. P.Premchand, "Breaking the Boundaries for Software Component Reuse Technology", International Journal of Computer Applications, vol. 13, pp. 37-41.
[5] A. Stierand, P. Reinkemeiert and T. G. t, P. Bhaduri, "Real-Time Scheduling Interfaces and Contracts for the Design of Distributed Embedded Systems", Proceedings of 8th IEEE International Symposium on Industrial Embedded Systems, **(2013)**, pp. 130-139.
[6] D. B. Stewart, R. A. Volpe and P. K. Khosla, "Design of dynamically reconfigurable real-time software using port-based objects", IEEE Trans., vol. 23, **(1997)**, pp. 759-776.
[7] M. Zeidler and B. Schulz Stelter, "Deliverable D6.2", PECOS Project Presentation, **(2002)**.
[8] R. Ommering, F. Linden and J. Kramer, "theKoala Component Model for Consumer Electronics Software", IEEE Computer, vol. 33, **(2000)**, pp. 78-75.
A. P. Ghosh Hazra and S. G. Vadlamudi, "Formal Methods for Early Analysis of Functional Reliability in Component-Based Embedded Applications", Embedded Systems Letters, vol. 5, **(2013)**, pp. 8-11.
[9] J. Suryadevara, E.-Y. Kang, C. Seceleanu and P. Pettersson, "Bridging the Semantic Gap Between Abstract Models of Embedded Systems", Proceedings of Component-based Software Engineering, ACM Sigsoft Symposium, **(1991)**, pp. 55-73.
[10] A. Vulgarakis, J. Suryadevara, J. Carlson, C. CerschiSeceleanu and P. Pettersson, "Formal Semantics of the Procom Real-Time Component Model", Proceedings of 35th Euromicro Conference on Software Engineering and Advanced Applications, Patras, Greece, **(2000)** August 27-29.
[11] Q. Wu and Y. Li, "An Adaptive and Semantic Component Model for Adaptive Middleware in Ubiquitous Computing Environments", Proceedings of Second International Workshop on Computer Science and Engineering, Qingdao, China, **(2009)** October 28-30.

Authors

Ruiqin Sun, was born in shandong, China, in 1989.She is currently the MS degree in Hunan University, Changsha, China. She received herBS degree inCollege of Computer Science and Electronic Engineering from Hunan University in 2008.Her current research interests include embedded systems and software product line.

Guanzhong Yang, was born in Changsha, China, in 1963, Professor Supervisor. From 1988 till now, he has been teaching in College of Computer Science and Electronic Engineering of Hunan University. His research interests include software product line, multimedia communication technology and application, network education technology and application, network quality of service and application research.

Dafang Zhang, was born in Shanghai, China, in 1959, Professor and Ph.D. supervisor. He is a professor at the College of Computer Science and Electronic Engineering,Hunan University. His main research interests include dependable and fault tolerant computing, network test and software fault tolerance.



Bowen Yang, was born in Changsha, China, in 1992.He received the BS degree in Central South University on 2014, and now studying in University of Ottawa majoring Electrical and Computer Engineering. He got the bronze medal in provincial ACM competition on 2015, and got the Silver medal in the next year. The topic of graduation thesis is Design of Test Method Based on Texture and Super pixel.