Software IC Revised: A New Approach of Component-Based Software Design with Software Slots

Shangwei Duan and Xiaobu Yuan
School of Computer Science, University of Windsor, Canada
swduan@cs.uwindsor.ca, xyuan@uwindsor.ca

Abstract

By investigating the failure of Software IC in object-oriented technology and studying the characteristics of component-based or COTS-based software technologies, this paper revises Software IC and develops a software-slot-oriented strategy for the design of component-based software systems. This new strategy introduces requirements directly into the design of component-based systems, and allows architects to focus on system frameworks without too much concern about components. This work provides a mechanism to deal with the increasingly complex interface logics of COTS systems, and suggests a possible guidance for component standardization. Effectiveness and availability of the proposed approach are illustrated with a case study that applies the software-slot-oriented design on a practical component-based software system.

1 Introduction

Component-based software technology (CBST) is a new software paradigm that has attracted huge attention from both the academic and industrial communities. By producing application systems with pre-constructed software pieces, CBST promises the benefits of accelerated software development, reduced costs, higher reusability, and greater flexibility [1]. In responding to the great potentials of CBST, several commercial off-the-shelf (COTS) products for the construction of component-based software systems have been introduced to the market [2]. These products include Enterprise JavaBeans (EJB) by Sun Microsystems Inc. and Common Object Request Broker Architecture (CORBA) by the Object Management Group. At the mean time, Microsoft Corporation put forward its products of Component Object Model (COM), Distributed Component Object Model (DCOM) and .NET.

However, many fundamental issues of CBST remain unsolved, from both the theoretical and practical points of view. In addition to the unjustified borrowing of ideas from the existing object-oriented (OO) technology in the practice of component-based software development, there has been no consensus even regarding the definition of components [3]. Different component infrastructures, as introduced with EJB, COM/DCOM/.NET, CORBA, and Web Services by different companies or groups, add further confusion to CBST. Although there are interoperations among these infrastructures, there is no sign of a united component infrastructure, at least in the near future. Despite the limited success of extensions to the United Model Language (UML) for components modeling, visually modeling of component-based software systems with different component infrastructure is one of the most difficult and challenging subjects in Component-based Software Engineering.

In principle, a component infrastructure consists of three models: a component model, a connection model, and a deployment model [3]. Among the three models, the component model defines characters of a standalone component, the connection model defines the ways how components are integrated, and the deployment model describes the methods how an existing component-based system works in a practical working environment. In comparison with the other two models, the connection model is more important as components are pre-developed in COTS systems. In addition, a specific working environment is beyond the scope of model design. The focus of this paper is therefore on the connection model for COTS software systems.

Existing component models treat components as either abstract entities, such as class, or concrete entities, such as objects. For EJB and .NET, components evolve directly from OO technology. They are more like special classes that have abstract descriptions and can be instantiated later. In other models such as Web Services and TinyOS, components work as concrete objects and subsystems without instantiation [4] [5]. It becomes more adequate for connection models to model components as concrete entities when different instances of the same abstract entities may communicate and interacts. These instances differ in their internal statuses and external configurations. As a result, connection models need also to have the mechanism to represent these components and their interactions.

COTS components are usually developed by third party vendors and released in binary codes. They work as black boxes. However, inheritance is referred as the white boxes adaptation [6]. Inheritance is not suitable for COTS-based system modeling because it requires system designers to understand the internal implementations of inherited components. To complement the loss of reusability without inheritance, COTS systems support wrapping interfaces and more flexible configuration [6]. In such a sense, reusability in component-based software systems is at a higher and coarser level than in OO systems. CBST helps to greatly reduce the complexity and flexibility of software systems with the sacrifice of certain level of reusability.

In light of the difficulty of providing visualization tools to support different models of component-based software development and based upon the observations of CBST, especially COTS, characteristics, this paper investigates the application of Software IC in CBST. The remaining of the paper is structured in such a way that, after a brief survey of prior works in Section 2, Section 3 first discusses the introduction of three new notations to revise Software IC for component-based technology. It then proposes a new strategy of component-based software design with software

Sixth International IEEE Conference on Commercial-off-the-Shelf (COTS)-Based Software Systems (ICCBSS'07)
0-7695-2785-X/07 $20.00 © 2007 IEEE

slots, and describes the steps involved in the designing process. Afterwards, Section 4 thoroughly studies the proposed notations with a practical component-based system. Conclusions and future works are presented at the end of the paper.

2 Related Works

This section highlights current work on component-based software visualization model and prior work on Software IC as they motivate and shape up the work of this paper.

2.1 Component-based Software Visualization

In general, there are two types of modeling methods for the representation of a software system, i.e., visualization models and symbolic models. A visualization model consists of self-explanatory graphical notations, which are easy for human beings to understand but difficulty for computers to process. In comparison, symbolic models have well-defined properties for computerized processing, and therefore are ready to be integrated into the process of components discovery and selection. Symbolic languages, however, are difficult for human beings to understand. As a result, software designers tend to choose a visualization model in the design phase, and use a symbolic model as an auxiliary tool in practice.

Due to the current practice in software industry, most of the work on component-based software visualization has been focused on extending UML for the use with component models. Savino-Vazquez and R. Ruigjaner presented a domain extension, termed DExOM/UML, as a formalism of HOOMA to represent both the architecture and interaction aspects of a component-based system [7]. Lee worked on the effectiveness of component modeling, and extended UML component diagrams with added message flows and classes [8]. J. Grundy and R. Patel extended UML to facilitate aspect-oriented component design, and used EJB to implement these designs [9]. To keep up with demands while maintaining the uniformities of UML, the latest version UML 2.0 officially proposed a component model as a specialized class model that has an external specification in the form of one or more provided and required interfaces [10].

The revised specification of UML and its extensions provide features to model some aspects of components and frameworks. However, they suffer from the problems of vague semantics in the construction of components, and overlapped semantics in classifiers, such as components, classes, and subsystems. The distinction between classes and objects are well known in OO technology, but it is not clear yet in CBST. It is unclear, for instance, components should be instantiated or not. UML 2.0 obviously treats components as classes. It works very well for components evolved from classes, but it is not suitable for such components as subsystems that may not be instantiated and do not support inheritance. This dilemma was observed by Cheesman and Daniels [11]. When they were creating a diagram of component object architecture to specify the relationship and interactions among instances of class-like components or subsystems-like components, the UML diagram behaved more like the variations of class diagram. Moreover, abstract class components must not appear in the diagrams of component object architecture, but this point was omitted regrettably. The diagram they created was finally rejected by UML 2.0 as an invalid diagram.

There are also a few efforts to work on component-based software visualization models outside the UML campus. CORBA Component Model (CCM), for example, proposed four types of ports: “facets”, “receptacles”, “event source”, and “event sinks” [12]. It uses facets and receptacles to describe provided and required interfaces, and uses event sources and event sinks to handle asynchronous communications between two components. Other types of connections are also possible, including the connections from a “facet” to a “receptacle” and from an “event source” to an “event sink”. Port connections work well in CCM for the modeling of component communication.

Nevertheless, UML, current UML extensions, and the other software visualization models, including CCM, fail to model the basic characteristics of component-based systems, especially in the construction of COTS-based systems. These models work only on the architecture of component-based system with class-like diagrams. Although components follow objects’ principle of integrating functions and related data, they are actually different from objects. The most important difference is that components are exchangeable when laid out in component-based systems. Component-based systems should make managing changes as the first concern [11]. Difference in reusability is also important, but it is not as important as exchangeability because component-based systems are built with pre-manufactured components. The designers of component-based systems care about the use of components, instead of the development of actual components. This is especially true for COTS-based systems.

When dealing with the abstraction of exchangeability, the concept of software contracts stands out as a good candidate for exchangeability representation. Software contracts define the rules and functionalities that all components in a system need to obey and provide. In other words, any component can be replaced in the system if its replacement obeys the same rules and functionalities. The concept of software contracts has been applied to the component world by Catalysis [13] in 1999 and further developed in the book by Cheesman and Daniels [11]. Their work demonstrated the great impact that software contracts have on component-based software systems, but none of them came close to the application of software contracts in the modeling of component-based software system design. Without new notations, the modeling of general components with software contracts remains at the conceptual level. Proper legends have to be created as part of the visualization model for component-based software systems, which leads to the investigation into Software IC.

2.2 Software IC

The term of Software IC stands for software integrated circuit. It was originally proposed as a design model to build reusable software components for OO software systems [14]. In its original introduction, a software IC is composed of a set of reusable objects in format of black boxes that are interweaved according to the specific connection standards. The fact that objects communicate by message passing plays a key role in Software IC. Motivated by the success of reusability concept in hardware integrated circuits, Software IC tries to minimize the cost of software development by promoting the reusability of objects in object-oriented design.

After the conceptual introduction of Software IC, several software researchers investigated details of its application in OO software development. Fagerstrom developed a control unit to build a software IC structural model to describe the logical relationships between components in a distributed system [15]. Chen and Sobkiw extended the concept of Software IC by suggesting object binding to support system reusability [16]. Lin even defined software pins and a specification language to describe relationship in the design of Software IC [17]. Software IC library and TLMP are also developed as implementation of this model [18].

Further investigation soon uncovered a fatal problem of Software IC in OO software development, i.e., the so-called fragile base class problem [19]. This problem limits the correctness of Software IC to only those cases when software interfaces can be defined as crisply as hardware interfaces [20]. The main reason behind this problem is that the fundamental encapsulation of circuit chips breaks down in OO software due to inheritance. An integrated circuit board is flat with all chips at the same level on the board, thus any changes of internal implementation of one chip will not promulgate to other chips. However, inheritance in OO places software objects at multiple levels, and changes of internal implementation at higher levels (super classes) may be transferred downstream to the lower levels (subclasses). It is a paradox that inheritance, as originally designed to introduce reusability, becomes the main reason that put an end to the development of a model that promotes reusability -- Software IC. After the discovery of the fragile base class problem, the term of Software IC disappeared almost completely from the literature.

One other important reason for the failure of Software IC in OO technology is the confusion between classes and objects [21]. Similar to hardware chips and integrated circuits, software ICs should not be meta-products but final products. It means that the basic pieces in a software IC model are concrete entities, more like the instances of classes. Although the relationships between classes are easier to represent than objects as the number of objects is much greater and their relationships are more complex, the confusion of classes and objects in Software IC fails to capture the characteristics of both hardware and software. Besides, there is also another important reason, i.e., the complexity and flexibility of OO systems. As an OO system normally consists of hundreds of objects and thousands of interactions, the varieties of objects and relationships among them are beyond the representative capability of integrated circuit.

3 Component-Based Software Design with Revised Software IC

This section revises Software IC with three new graphic notations, and develops a software-slot-oriented strategy to help software architects designing the architecture of component-based software systems by matching components with user requirements.

3.1 Revision of Software IC

Encapsulation is fundamental in Software IC. As a software IC specifies a border around an arbitrary set of processes, objects, or routines, it is referred to by the border and name associated with it. The effluences of processes, objects, and routines are limited within the scope of a particular software IC, which is similar to the mechanism of name space in OO technology. To apply the concept of Software IC, this paper considers a software IC as a static, language-independent software entity with no instantiation or inheritance. A software IC consists of a set of independent processes, objects, or routines encapsulated in a black box. To revise Software IC and allow component-based software architects and developers to create illustrative diagrams, the following discussions introduce three new graphic notations.

These three new notations are software ICs, software pins, and software slots. In analogue to pins of an electronic chip, a software IC uses software pins for input and output operations. A software pin models the interface prototypes of a software IC. Correspondingly, a typical software pin has three characteristics. The first characteristic is Pin ID that gives the name or identification of a pin, and the second is Data Stream that models data conveyed across software pins between two software ICs. The data stream could be simple data, such as integer and characters, or complex data, such as structured records or multimedia data. Figure 1 illustrates a typical software IC with different types of software pins.

The last one, which is also the most important, is Pin type. A Pin type classifies a pin to be either an attribute pin or an interface pin. Attribute pins model public properties of software ICs, and are divided into two sub-types as In-attribute and Out-attribute. The former is for configuration, and the latter is for query of internal status. In comparison, Interface pins model the external behaviors of software ICs, and define their operations as well. According to operation direction and the way to coordinate two software ICs, Interfaces pins are further divided into four sub-types that describe different communications. They are In-Sync, Out-Sync, In-Async, and Out-Async. The functionalities of software pins are listed in Table 1.

3.2 Software Slots

A software IC is in effect only when it interacts with other software ICs through software slots in the system. A software slot is a special entity responsible for linking and un-linking software ICs, and it abstracts and models system framework in the format of software pins. A software slot connects software ICs by their interface pins. Similar to software interface pins, software slots also have four sub-types: In-SyncSlot, Out-SyncSlot, In-AsyncSlot, and Out-AsyncSlot. The functionality of slots and the relationships of slots and pins are listed in Table 2. Figure 2 illustrates a diagram of software slots linking two software ICs.

The multiplicity of association relationship of In-SyncSlot (In-AsyncSlot) and Out-SyncSlot (Out-AsyncSlot) is one to many or many to one. In this revised Software IC, if one pin inserts into an In-SyncSlot (In-AsyncSlot), at least another pin must insert into its corresponding Out-SyncSlots (Out-AsyncSlots). This restriction corresponds to the fact that if one interface is used, this interface must be implemented at least once or vice versa in a practical system. Furthermore, the pins inserted into the corresponding slots of one slot could be from different software ICs. This mechanism provides a way to achieve polymorphism of software ICs in component-based software technology. In other words, although an Out-Sync (Out-Aysnc) software pin of a software IC can only be inserted into one slot, it may have multiple corresponding slots, thus have multiple corresponding In-Sync (In-Aysnc) pins on the other side. It means that an interface used by a component could be implemented by two or more

components based on different data streams or configuration of software ICs.

Table 1: Functionalities of software pins

Type	Sub-type	Functionality
Attribute	In	Configures public parameters of software ICs to adjust their behaviors and performance.
	Out	Outputs the internal status of software ICs to monitor their behaviors and performance.
Interface	In-Sync	Methods implemented by software IC and they return the value only after they complete.
	Out-Sync	Methods used by software ICs and they receive the return value only after their In-Sync counterparts complete.
	In-Asyn	Events implemented by software ICs without return value.
	Out-Async	Events used by software ICs without return value.

Table 2: software slots and software pins

type	Slot	Pin	Functionality
1	In-SyncSlot	In-Sync	Model providing methods
2	Out-SyncSlot	Out-Sync	Model using commands
3	In-AsyncSlot	In-Async	Model providing events
4	Out-AsyncSlot	Out-Async	Model using events

3.3 Multi-level Software IC

The encapsulation characteristic of software slots allows the design of multi-level software ICs, in which two or more software ICs can be grouped together though software slots or one software IC is wrapped with additional codes. A multi-level software IC works as a new software IC from an external view. This mechanism empowers the revised Software IC with higher level of reusability, and provides flexible usability of software ICs. The diagram of a multi-levels software IC is illustrated in Figure 3. Multi-level software IC does not break the encapsulation because all software ICs in one diagram only interact with each other at the same level. The internal software ICs, slots and additional codes are transparent to the external software ICs and slots. Actually, the use of multi-level software ICs increases the usability of software ICs, which is further discussed in Section 3.5.

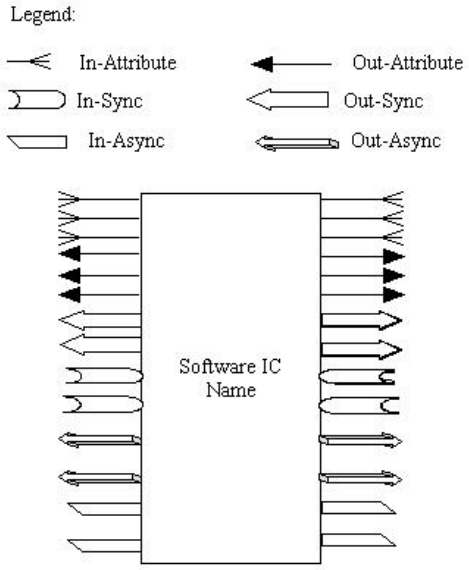

Figure 1: Software IC

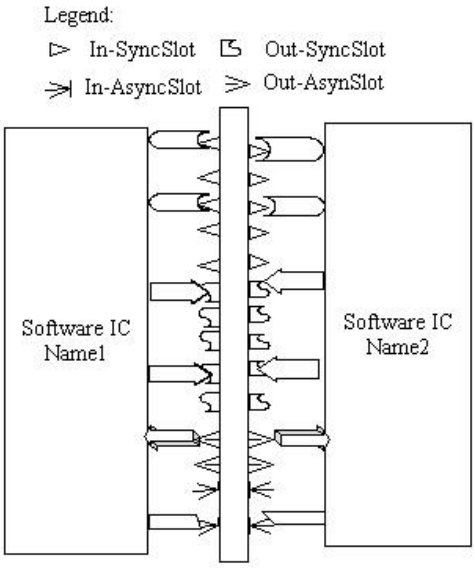

Figure 2. Software Slots linking two Software ICs

3.4 The Process of Software Slot Oriented Design

Supported by notations of the revised Software IC, software design based on software slots takes the following five steps.

(1) Select appropriate component-based technologies. If the chosen technology cannot avoid inheritance, architects should resort to another choice.

(2) At the architecture level, design or select software slots from the software slots library, with the support of multiplicity relationships for In-SyncSlot and Out-SyncSlot, or In-AsyncSlot and Out-AsyncSlot.

(3) At the development level, select software IC (components) from software IC (components) libraries based on the

selected software slots in the previous step if they are available, and develop them otherwise.

(4) Design multi-level software ICs, if possible, to use a group of components as a whole or add extra functionalities for a component. It is noted that all class-like components, no matter if they are selected or designed, must be instantiated before use.

(5) Link software ICs with software slots to accomplish anticipated interactions among software ICs in the system.

In the designing process, the design of software slots in the second step is the key in an interface-oriented design, which gives the name of this process as software-slot-oriented design. The framework of a system can be established only after the design of software slots. Software slots could be designed by software architects or selected from standard slot libraries. At the beginning stage of a component-based technology, most of software slots are designed by architects according to the functionalities of requirements. When the development of component-based technology reaches more advanced stages and when standardization of slots is mature, more software slots will come from standard libraries.

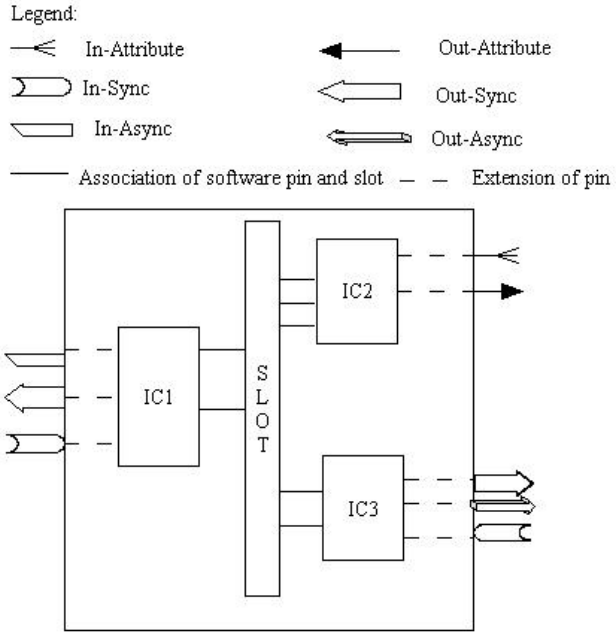

Figure 3: Multi-levels Software IC

3.5 Discussions

The new notations introduced with the revised Software IC are simple in illustration, yet they capture well the characteristics of component-based development in all the aspects as listed below.

- Procurement-centric. It is a big shift from development centric in object-oriented development to procurement-centric in component-based development. This shift makes the selection/integration of components or COTS a distinguished phase of component-based development. The class diagrams and component architecture diagrams of UML 2.0 cannot cover this activity of component-based development. Nevertheless, the revised Software IC provides software slots to model system framework in analysis, and software ICs and software pins to model components and their interfaces in implementation. The selection/integration of components or COTS becomes a unique phase of the software-slot-oriented design process that uses the plugging/unplugging of software pins into/out from software slots to model the matching of components and requirements. A qualified software IC must have at least one software pin for it be plugged into a software slot, and the linked software ICs must have at least one software pin matching with the software pin of this software IC.

- Exchangeability. Exchangeability is the most important in component-based or COTS-based systems. In the revised Software IC, a software IC can be replaced by any other software IC(s) as long as the replacing software IC(s) has (have) software pins to cover all the useful software pins of the replaced software IC. A useful software pin refers to a software pin being inserted into a software slot in the system, and a software pin is replaceable when it has the same functionalities as another software pin. This exchangeability is partly expressed in UML component diagrams with its ports, connectors, and interfaces, but this expression ability is greatly enhanced with the help of software slots.

- Standardization. The use of software slots makes it possible to standardize component interfaces with software pins. In a component-based system, most of the functionalities are either shared within the infrastructure of a component technology family or common with other component technologies. Those are the candidates that should be standardized at the family or universal level. The revised Software IC provides the tool for standardization as standardized software slots can be used to models standardized functionalities. Software pins can also be standardized accordingly based on their corresponding software slots.

- Configuration. Configuration presents a special meaning in component-based or COTS-based systems when components work as black boxes that cannot be optimized in the code level or internal mechanism [22]. In addition, the variations of component-based system are made by means of different configurations [13]. While UML 2.0 ignores the configuration issue of components, the revised Software IC provides a mechanism to deal with this issue with attribute software pins.

- Reusability and usability. As discussed in Section 1, in comparison to objects, components lose some reusability for the coarser and higher granularity. It is also worthwhile to point out that components actually have higher usability than objects for the same reason. Reusability and usability form a contradictory pair, and there is currently no optimal granularity degree that balances both the reusability and usability for a given system. It is the high complexity of software systems that makes it extremely difficult for any attempt to quantitatively calculate such a degree. In the revised Software IC, however, it becomes feasible to provide a method for component-based software designers to qualitatively achieve such a degree by manually balancing the reusability and usability for one or a group of components in a system. The mechanism of multi-level software ICs permits software ICs and slots to form a greater software IC when a group of components needs to increase

its usability. The usability of a component can also be increased by wrapping with additional codes. Increased reusability is achieved when software pins of a software IC are used separately.

There is a fundamental difference between the current UML 2.0 and the revised Software IC. While the former focuses on the relationships among components, the latter focuses on the relationships between requirements and components. The revised Software IC has no intension to either redesign the comprehensive modeling ability of UML or replace UML in component-based system design. Instead, it uses software slots to separate the work of software design into the work on framework and components, thus transferring the task of architects from designing a whole system to designing software slots. It also allows component-based software developers to concentrate working on software development by selecting appropriate software ICs and software IC pins for implementation.

4 Case Study

This section presents a case study, which goes through in detail the process of software-slot-oriented design with a component-based software system, called Surge.

4.1 System Overview and Requirements

Surge is a network application whose local hosts collect data, such as light strength and temperature, from wireless sensor nodes located at different locations. At the same time, the system receives commands from local hosts, and broadcasts them within the entire wireless sensor networks. Surge is developed under the infrastructure of tinyOS [5], and tinyOS is a component-based embedded and real-time operating system developed by University of California at Berkeley. In tinyOS, components work as subsystems and cannot be instantiated and inherited. Listed in Table 3 is a set of ten requirements of the Surge system, in which the first eight are functional and the last two are non-functional requirements.

Table 3: Requirements List

R1	Each wireless node samples data from its sensor periodically according to the pre-defined interval.
R2	Raw data is conveyed to a based node that connects to a local host with wired lines.
R3	Each wireless node, except the central node, works as switch node to transfer data from one of its neighbors to another neighbor who is responsible for re-transfer the data to the base node.
R4	Each wireless node maintains a routing table to switch data and update this table in case of topology changes. Through the routing table, all nodes in the Surge system form a routing tree rooted in the based node.
R5	Each wireless node receives the commands from the local host to broadcasts to its neighbors.

Table 3: Table 3: Requirements List (Cont')

R6	Each wireless node updates its parameters according to the commands at run time.
R7	Each wireless node initializes and starts its task by itself when it is powered on.
R8	The running status is displayed by Led light.
R9	Since wireless nodes are powered by batteries, a reasonable sampling rate and transmitting rate need to be carefully selected to reduce energy consumption and maintain satisfying accuracy.
R10	Because resources, memory, and CPU ability in each node are highly limited, the routing table in each node needs to be small enough while providing an efficient route to the base.

4.2 Software Slots

As the Surge system does not involve inheritance, the proposed software-slot-oriented process is applicable to its design. Corresponding to the ten requirements given in Table 3, software slots are defined at the architecture level, and listed in Table 4. In the table, there is no difference between in-slots and out-slots because any slot has in and out features at the same time for the Surge system. The last column of Table 4 matches the designed software slots with the original requirement(s).

Table 4: Software Slots of Surge

Group	Slot	type	Functionality	R
ADC	Get-Data	Sync-Slot	Retrieve light data from sensor	R1
	Data-Ready	Async-Slot	Indicate light data are ready	R1
Route Control	Get-Parent	Sync-Slot	Get the node's parent address	R4
	Get-Depth	Sync-Slot	Get the node's depth in the routing tree	R4
	Get-Sender	Sync-Slot	Get the previous hop sender	R4
	Set-Update-Interval	Sync-Slot	Get the update period of a routing table	R4
	Manual-Update	Sync-Slot	Update the routing table when necessary	R4, 10
Send	Send	Sync-Slot	Transmit data to the node's neighbor	R2,3
	Send-Done	Async-Slot	Indicate sending completeness	R2,3

Table 4: Software Slots of Surge (Cont')

Route Select	isActive	Sync-Slot	Indicate a valid route	R4
	Select-Route	Sync-Slot	Select a route and fill such info. to a packet	R4
	Init-Fields	Sync-Slot	Initialize a entry of the routing table	R4
Group	Slot	type	Functionality	R
Receive	Receive	Async-Slot	Indicate the address of received data buffer	R2,3,5,6
Timer	Start	Sync-Slot	Start timer	R1,9
	Stop	Sync-Slot	Stop timer	R1,9
	Fired	Async-Slot	Indicate if the timer triggers an event.	R1,9
Led	Init	Sync-Slot	Initialize Led	R8
	Ledon	Sync-Slot	Power on Led	R8
	Ledoff	Sync-Slot	Power off Led	R8
	Led-toggle	Sync-Slot	Toggle Led	R8
Std-Control	Init	Sync-Slot	Initialize component	R7
	Start	Sync-Slot	Start component	R7
	Stop	Sync-Slot	Stop component	R7

4.3 Software IC

At the development level, five software ICs are selected from the tinyOS components library to fulfill the functionalities defined in the software slots. They are Photo, Bcast, Ledc, TimerC, and MultiHopRouter. A new software IC, namely SurgeM, is then developed to build a new system with the five selected software ICs. SurgeM works as the central software IC in the system. Each of its interface in-pins (out-pins) has at least one out-pin (or in-pin) belonging to the five software ICs in the component library. In addition, it defines two In-Attribute pins, samlpingRate and transmisionRate, to fine tune system performance.

Details of the selected and designed software ICs are listed in Table 5. It is noted that every software IC has StdControl software Pins, which has three In-SynPins, i.e., init, start, and stop. StdControl takes similar format as the other software ICs, and therefore is not listed in the table to reduce table length for concise illustration.

Table 5: Software ICs in Surge System

Name	Pins	Type	Group
Photo	getData	In-SyncPin	ADC
	dataReady	Out-AsyncPin	
Bcast	receive	Out-AyncPin	Receive
Ledc	ledon	In-SyncPin	Led
	Ledoff	In-SyncPin	
	ledtoggle	In-SyncPin	
TimerC	start	In-SyncPin	Timer
	stop	In-SyncPin	
	fire	Out-AsyncPin	
Multi Hop Router	Send	In-SyncPin	Send
	Receive	Out-AsyncPin	Receive
	getParent	In-SyncPin	Router Control
	getDepth	In-SyncPin	
	getSender	In-SyncPin	
	setUpdateInteval	In-SyncPin	
	manualUpdate	In-SyncPin	
SurgeM	getData	Out-SyncPin	ADC
	dataReady	In-AsyncPin	
	receive	In -AyncPin	Receive
	ledon	Out -SyncPin	Led
	Ledoff	Out -SyncPin	
	ledtoggle	Out -SyncPin	
	start	Out -SyncPin	Timer
	stop	Out -SyncPin	
	fire	In -AsyncPin	
	Send	Out -SyncPin	Send
	Receive	In -AsyncPin	Receive
	getParent	Out -SyncPin	Router Control
	getDepth	Out -SyncPin	
	getSender	Out -SyncPin	
	setUpdateInteval	Out -SyncPin	
	manualUpdate	Out -SyncPin	
	samlpingRate	In-Attribute	
	transmisionRate	In-Attribute	

4.4 Multi-level Software IC

Among the six software ICs, MultiHopRouter manages the routing table, which is the essential data structure of the Surge system. Due to the complexity of routing protocol and the fact that the tinyOS Component library does not have existing components

to use, MultiHopRouter is designed as multi-level software IC. It consists of five smaller components that comes directly from tinyOS component library and connects through pre-defined software slots. The internal structure of multi-level software IC is illustrated in Figure 5.

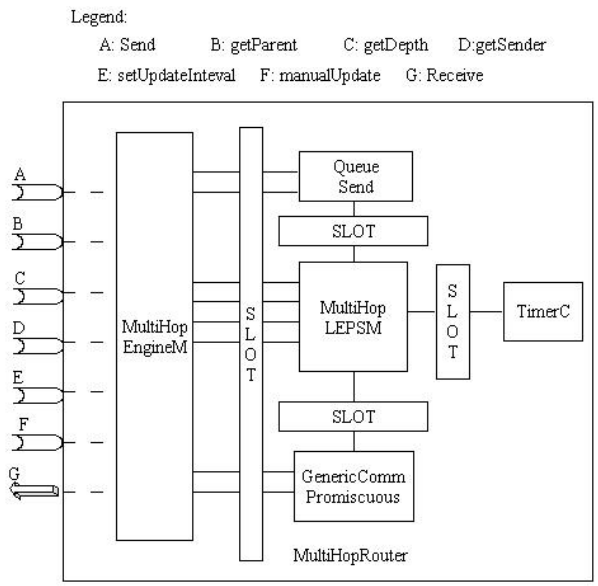

Figure 5: The Internal Structure of MultiHopRouter

4.5 System Establishment

After the selection and design of software slots and software ICs in the previous steps, the software ICs can then be connected with software slots to fulfill the functionality of the system. A design diagram is illustrated in the Figure 6.

Similar to the practice of Printed Circuit Board (PCB), the design diagram of Surge clearly describes the components. In addition, the construction of software ICs and software slots matches with system requirements. Further adjustment to the behavior and performance of Surge can be made by configuring two In-Attribute software pins: samplingRate and transmissionRate. As illustrated in Figure 6 and described in Table 5, TimerC and LedC are two reusable components that can be used in any wireless sensor networks application that needs a timer and shows the internal status with a Led light. Meanwhile, if another timer component is developed, it can replace TimerC as long as they have the same software pins. Therefore, the revised Software IC explicitly indicates the reusability and exchangeability of the software system under construction.

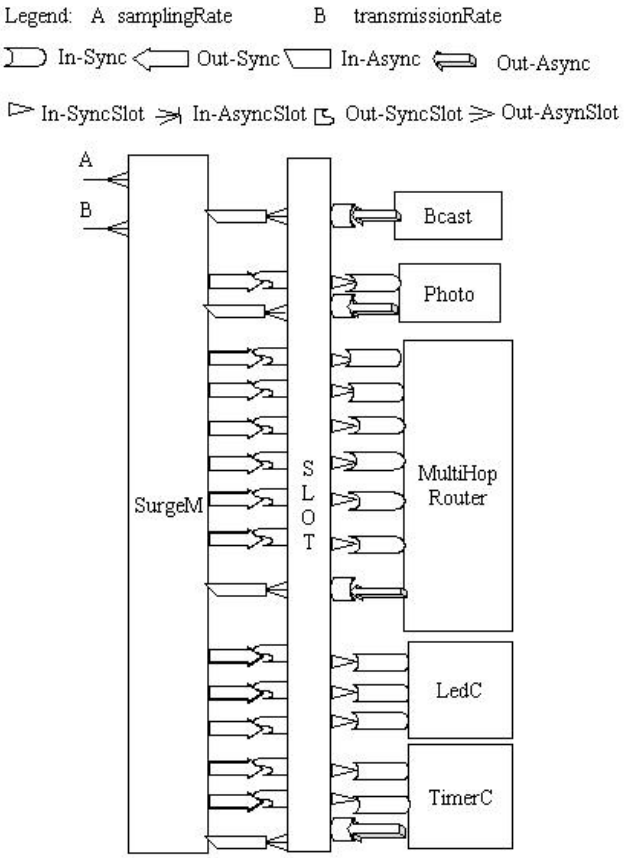

Figure 6. Surge System

5 Conclusions and Future Works

This paper introduces Software IC to the design of component-based software systems. By revising Software IC with three new visualization notations, it develops a new approach of designing component-based software systems with software slots. In comparison to UML 2.0, the proposed software-slot-oriented design strategy captures the characteristics of component-based or COTS-based technologies. The use of software slots introduces requirements directly into the design of component-based system, and allows architects to focus on the system frameworks without too much concern about components themselves. The efficiency and applicability of the proposed design strategy are illustrated with a case study of a practical component-based system.

Instantiation and inheritance imposes limits to Software IC. As component-based technologies allow the initiation of components before being introduced to software design, it does not really limit the applicability of Software IC in component-based software development. Inheritance, however, breaks down encapsulation, which is fundamental to the revised Software IC. There is no problem to use the revised Software IC for the development of software based on the popular service-oriented architecture (SOA) as SOA does not support inheritance between services. When applying the revised Software IC in the design of component-based software systems, inheritance will have to be limited within the scope of components.

Although the revised Software IC does not support inheritance and instantiation, it can be extended with more notations to

describe component-based software design with more accuracy. Software IC, however, cannot describe all features involved in the design of component-based systems, such as state machine, time sequences, etc. As UML is being upgraded to support the practice of component-based technologies, the revised Software IC provides a complimentary set of notations to the new UML. A combination with the multiple notations of UML will help to overcome the deficiency of the revised Software IC.

6 Acknowledgments

This work was supported by Natural Sciences and Engineering Research Council (NSERC) of Canada.

References

[1] C. Szyperski, Component software: Beyond Object-Oriented Programming, 2nd ed. Addison-Wesley, 2002.

[2] Heineman, George T., Councill, William T., Component-Based Software Engineering: Putting the Pieces Together, Addison-Wesley，2001.

[3] A. Wang, K. Qian, Component-Oriented Programming, 1st ed., Wiley-Interscience, 2005.

[4] S. Chatterjee, J. Webber, Developing Enterprise Web Services: An Architect's Guide, Prentice-Hall, 1st Edition, 2003.

[5] P. Levis, S. Madden, J. Polastre, R. Szewczyk, K. Whitehouse, A. Woo, D. Gay, J. Hill, M. Welsh, E. Brewer, and D. Culler, TinyOS: An Operating System for Sensor Networks, in Ambient Intelligence, Page(s): 115–148, 2005.

[6] J. Mykkänen, M. Sormunen, K. Karvinen, T. Tikkanen, S. Päiväniemi, Component and Service Technology Families, Studies and Reports of the PLUGIT project 1, Kuopio, 2004.

[7] N.-N. Savino-Vazquez, R. Ruigjaner, A UML-based method to specify the structural component of simulation-based queuing network performance models, In Thirty-Second Annual Simulation Symposium, 1999, Page(s): 71-78.

[8] S. Lee, Y. Yang, F. Cho, S. Kim, S. Rhew, COMO: a UML-based component development methodology, In Proceedings of Sixth Software Engineering Conference (APSEC '99), 1999, Page(s): 54 – 61.

[9] J. Grundy and R. Patel, Developing Software Components with the UML, Enterprise Java Beans and Aspects, 13th Australian Software Engineering Conference (ASWEC'01), Australian,27-28 Aug. 2001 Page(s):127 - 136.

[10] OMG, Unified Modeling Language: Superstructure, http://www.omg.org/technology/documents/formal/uml.htm, 2004.

[11] J. Cheesman, J. Daniels, UML Components, A simple process for specifying component-based software, Addison-Wesley, 2001.

[12] R. Marvie, P. Merle, CORBA Component Model: Discussion and Use with OpenCCM, Technical report, Laboratoire d'Informatique Fondamentale de Lille (LIFL), 2001.

[13] D.F. D'Souza and A.C. Wills, Objects, Components, and Framework with UML: The Catalysis Approach, Addison-Wesley, 1999.

[14] L. Ledbetter, B. Cox, Software-ICs: A Plan for Building Reusable Software Components, Byte, 10(6): 307-316, 1985.

[15] J. Fagerstrom, Design and test of distributed applications, Proceedings of the 10th international conference on Software engineering, April 11-15, 1988, Singapore, Page(s): 88-92.

[16] T. L. Chen, W. Sobkiw, Binding as a Mechanism to Support Resuability in a Distributed Ada Communications System, In Proceedings of the sixth Washington Ada symposium on Ada table of contents, ACM Press, 1989, Page(s): 155-162.

[17] J. Lin, Cross-platform software reuse by functional integration approach, in Proceedings of COMPSAC '97, 1997, Page(s): 402-408.

[18] Solu Corp., TLMP: The software IC concept, Available at http://www.solucorp.qc.ca/tlmp/components.hc?webstep=5.

[19] J. Gosling, Java Intermediate Bytecodes, ACM SIGPLAN, Workshop on Intermediate Representations (IR '95), 1995, Page(s): 111-118.

[20] W. Tracz, Software Reuse Myths Revisited, International Conference on Software Engineering, In Proceedings of the 16th international conference on Software engineering (ICSE'94), 1994, Page(s): 272-273.

[21] C. Szyperski, Component software: Beyond Object-Oriented Programming, 2nd ed. Addison-Wesley, 2002.

[22] Xiaobu Yuan, Shangwei Duan, Zhiyong Liu, Explore robust component-based system, International Conference on Software Engineering, Proceedings of the 2006 ACM international workshop on Software quality, May 2006,Shanghai, Pages: 75-80.