CREAM: A Generic Build-time Component Framework for Distributed Embedded Systems

Chetan Raj[1], Jiyong Park[1], Jungkeun Park[2] and Seongsoo Hong[1]

[1]Real-Time Operating Systems Laboratory
Seoul National University, Seoul 151-744, Korea
{chetan, parkjy, sshong}@redwood.snu.ac.kr

[2]Dept. of Aerospace Information Engineering
Konkuk University, Seoul 143-701, Korea
parkjk@konkuk.ac.kr

Abstract

A component framework plays an important role in CBSD as it determines how software components are developed, packaged, assembled and deployed. A desirable component framework for developing diverse cross-domain embedded applications should meet such requirements as (1) lightweight on memory use, (2) integrated task execution model, (3) fast inter-component communication, (4) support for distributed processing, and (5) transparency from underlying communication middleware. Although current embedded system component frameworks address some of the above requirements, they fail to meet all of them taken together. We thus propose a new embedded system component framework called CREAM (Component-based Remote-communicating Embedded Application Model). It achieves these goals by using build-time code generation, explicit control of task creation and execution in the component framework, static analysis of component composition to generate efficient component binding, and abstraction of the component's application logic from the communication middleware. We have implemented the CREAM component framework and conducted a series of experiments to compare its performance characteristics to a raw socket-based communication implementation and the Lightweight-CCM implementation by MicoCCM.

Keywords: CBSD, Component Models, CCM, Koala, AUTOSAR, CORBA

1. Introduction

The ever increasing complexity of software has led to the wide adoption of component-based software development (CBSD) [1, 2]. The CBSD is an engineering methodology used to build a software system by composing software components. The CBSD requires less time to assemble components than to design, code, test and debug the entire system. This development methodology greatly reduces the software cost and the time to market.

In order for independently developed components to be seamlessly integrated with each other, there must be certain rules that govern how components are developed, packaged, assembled and deployed. The component framework enforces the component to adhere to these rules by providing gluing mechanisms for component composition, communication, synchronization, deployment and execution.

The current component frameworks for embedded systems have been designed based on existing enterprise computing component frameworks or from scratch to suit for a particular application domain. Popular embedded system component frameworks such as Lightweight-CCM [4], SCA [5] and .NET compact framework are designed based on existing enterprise computing component frameworks. However, they still require heavy resources and have significant performance overhead as they retain many of the fundamental features to guarantee the backward compatibility with their base component frameworks. For example, Lightweight-CCM is based on CCM and they both use the heavy CORBA [6] middleware.

There have been component frameworks designed from scratch for the embedded systems. Koala [7], AUTOSAR [8], and PECOS [9] are widely known examples. However, they are highly optimized for specific application domains and it is almost impossible to use them in other domains. For example, AUTOSAR uses domain-specific real-time control networks such as CAN and FlexRay. Therefore, AUTOSAR is not suitable for generic in-vehicle entertainment systems where those control networks are seldom used.

In this paper, we propose CREAM (Component-based Remote-communicating Embedded Application Model) as a generic build-time component framework for embedded systems. Specifically, CREAM is designed for the following five requirements essential for developing the current-generation of cross-domain embedded applications.

1. Lightweight on memory usage
2. Integrated task execution model
3. Fast inter-component communication
4. Support for distributed processing
5. Transparency from underlying communication middleware

To the best of our knowledge, CREAM is the only component framework that strives to achieve all the above design requirement taken together. The existing component frameworks meet only subsets of these requirements. For example, Koala and PECOS lack support for distributed processing. AUTOSAR is highly dependent on the OSEK-COM communication middleware. Lightweight-CCM and SCA require a significant amount of memory and CPU time.

The main idea of CREAM is to utilize build-time information and static analysis of the final component-composed system in order to improve the run-time performance and reduce the usage of system resources. Another main contribution of the CREAM component framework is the separation of the component model from the underlying communication middleware. This mechanism enables CREAM to support different communication middleware without modifying the component business-logic source code.

The remainder of this paper is organized as follows. In Section 2, we enumerate the design requirements for our component framework. In Section 3, we present the CREAM component framework along with its component model. In Section 4, we explain the key mechanisms used in CREAM to achieve the design requirements. In Section 5, we describe the CREAM implementation and experimental results. Finally, in Section 6, we provide our conclusions.

2. Design Requirements

The CREAM component framework strives to meet the following five design requirements that are essential for developing the current-generation of cross-domain embedded systems applications.

1. Lightweight on memory usage: Despite decreases in prices of solid state memory devices, memory is still a precious resource in embedded systems. Embedded system applications generally run on little memory.
2. Integrated task execution model: Embedded systems applications generally have many active components with independent threads of control. Moreover, many embedded systems applications have real-time constraints. In such systems, handling of task creation and execution forms an important activity. Explicitly controlling those activities in the component framework provides greater predictability and analyzability of the embedded systems applications.
3. Fast inter-component communication: Components can communicate with each other using various methods. If they are located in the same address space, a simple direct method call is sufficient. On the other hand, remote procedure call (RPC) should be used when components are in different address spaces or in different physical nodes. Therefore, a suitable communication mechanism must be chosen depending on components deployment location.
4. Support for distributed processing: Many embedded control systems such as automobile systems consist of tens of distributed nodes. Therefore, the support for distributed processing is becoming a prerequisite for an embedded system component framework.
5. Transparency from underlying communication middleware: A component framework useful for developing cross-domain applications should be independent of communication middleware and the underlying networks. For example, a networked home service robot having its own communication middleware needs to co-operate with home networked appliances using another communication middleware. Therefore, component construction and deployment should be transparent from the underlying middleware.

The support offered by existing embedded system component frameworks for these design requirements are as shown in Table 1.

Table 1. Comparisons of Component Frameworks

Component Framework	Light-weight on memory	Integ-rated task-exec-ution model	Fast inter-component communi-cation	Support for distributed Processing	Transparency from communi-cation middleware
Koala	**Yes**	**No**	**Yes**	**No**	**-NA-**
PECOS	**Yes**	**Yes**	**Yes**	**No**	**-NA-**
AUTOSAR	**Yes**	**Yes**	**Yes**	**Yes**	**No**
CCM	**No**	**No**	**No**	**Yes**	**No**
SCA	**No**	**No**	**No**	**Yes**	**No**
CREAM	**Yes**	**Yes**	**Yes**	**Yes**	**Yes**

3. The CREAM Component Framework

The CREAM component framework manages the underlying component model. It uses services of an object-based communication middleware to support remote inter-component communication. The CREAM component framework defines the component composition and deployment semantics. It makes use of XML based domain-profiles to describe, configure and deploy components in the final component-composed system.

3.1. Component Model of CREAM

The component model used in CREAM is similar to that of other popular component frameworks such as CCM and AUTOSAR. This component model can be visualized as shown in Figure 1 (a). A component interacts with other components and its environment using *ports* [2, 3].

3.1.1. Components' Port

A *port* is defined as a point of interaction between a component and its environment. These interactions occur through well-defined *interfaces* [2]. The ports in CREAM can be further categorized into client-server ports and event-service ports.

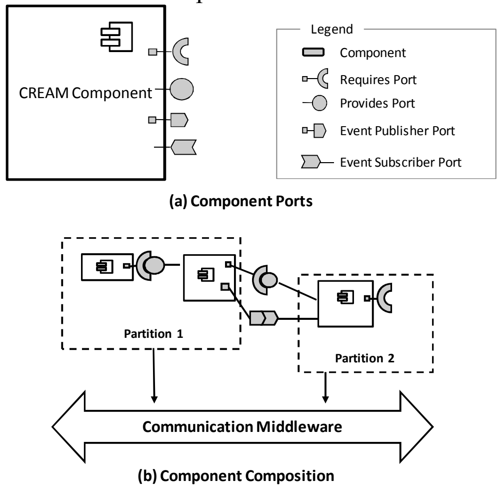


Figure 1: Component Model of CREAM.

(a) **Client-Server ports:** They represent synchronous communication between components. These ports have an interface type defined by the component developer. The server port is named as p*rovides* port. The client port is named as *requires* port. In CREAM, a *requires* port is an object reference that is associated with a *provides* port object instance of the same interface type.

(b) **Event-service ports:** They represent asynchronous interactions between components. Event ports are based on the *push-type* publisher-subscriber event model.

3.1.2. Interfaces

The interface of a port object is described using the CREAM's *interface definition language* (IDL). The CREAM makes use of a simple IDL supporting basic data types such as string, integer and floating point data types. The CREAM's IDL is transparently mapped to the IDL used by the underlying communication middleware for marshalling and un-marshalling of remote procedure calls (RPC).

3.2. Communication Middleware

The CREAM uses a lightweight communication middleware for supporting distributed processing. In general, any object-based communication middleware that supports marshalling and un-marshalling of object method calls can be used. The CREAM code generator can be extended to support any object-based communication middleware without requiring the costly re-coding of existing components' business logic.

3.3. Component Composition and Deployment

Component composition is defined as a process of integrating two or more components into a single unit. In CREAM, the composition of client-server ports involves associating *requires* port object references of one component with *provides* port object instances of another component. The event-service ports are composed together by associating event publisher and subscriber ports to a common event channel as accomplished in other *push-type* event models.

In CREAM, deploying components involves grouping of component instances into different *partitions*. A *partition* is executed as an OS process. All component instances of the same *partition* form collocated components and share the same address space. These *partitions* are managed by a separate standalone `DomainExecutionManager` which waits for the boot up of all *partitions*. It can then be used to start and stop the execution of *partitions* in the system.

3.4. Domain Profiles

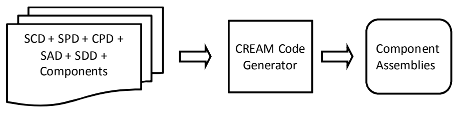


Figure 2: Domain Profiles Processing.

The CREAM component framework makes use of XML based domain profiles as its *component definition language* [2] for describing various operations on components. These domain profiles are – (1) Software Component Descriptor (SCD) used for specifying and developing components, (2) Software Packaging Descriptor (SPD) for describing the software component package, (3) Component Properties Descriptor (CPD) for describing the custom properties of component instances, (4) Software Assembly Descriptor (SAD) for composing components to form an assembly, and lastly (5) Software Deployment Descriptor (SDD) which provides the partitioning and deploying information. These domain profiles are consumed by the CREAM's code-generator to produce the final component assemblies as shown in Figure 2.

4. Key Mechanisms of CREAM

The key mechanisms of CREAM that achieve the aforementioned design requirements are explained in this section.

4.1. Build-Time Code Generation for Developing a Lightweight System

The CREAM is a build-time component framework. The component framework binds all component references and dependencies at build-time. This analysis helps remove costly memory consuming features such as XML-parsers, naming-services and dynamic component binding to achieve a lightweight system.

The CREAM code generator analyzes the domain profiles and extracts required information at build-time. This information includes the components' interfaces and ports, inter-connection of components' ports, custom properties of component instances, partition and deployment information. The code generator uses this information to generate statically configured code that instantiates the components, inter-connects the components' ports and deploys the composed components. This static analysis and build-time code generation removes the need for a heavy run-time and enables developing a lightweight final system.

4.2. Integrated Task Execution Model

Handling of task creation and execution forms an important activity in embedded software systems. These systems usually have many active elements that need their own threads of control. Manual coding of task creation and execution for such active elements causes the strong coupling of applications to target platforms. Moreover, manual coding for task creation leads to difficulties in predictability and analyzability of the embedded application system. To address this problem, the CREAM has integrated the task execution model into the component framework. The CREAM explicitly controls the creation and execution of all tasks in the system. This integrated task model enables automatic synchronization among shared component instances and helps analyze the WCET of tasks.

In CREAM, components are of two types: (1) *active* components, with an independent thread of control, and (2) *passive* components, with no independent thread of control. In CREAM, active components implement a `run` method. The CREAM component framework creates a task and initializes its entry point to the `run` method for each active component.

The task execution model in the CREAM component framework can be described in Figure 3. The CREAM framework in each *partition* creates a component service thread for all passive components. The framework then creates active run threads for each active component in the *partition*. All remote method invocation (RMI) on a method of a passive component is executed within the context of the component service thread. The inter-component communications between all components within a partition occur through simple local function calls. Each *partition* registers themselves with their network port and location details with the DomainExecutionManager which is then used to start and stop executions of all *partitions*.

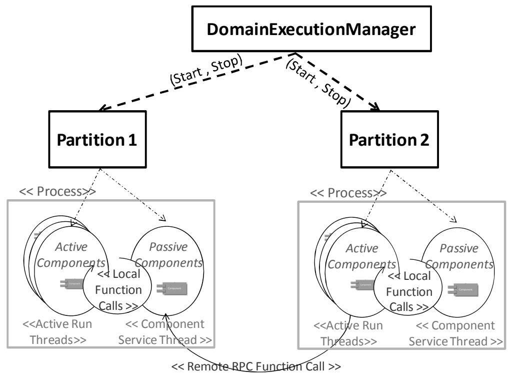


Figure 3: Task Execution Model of CREAM.

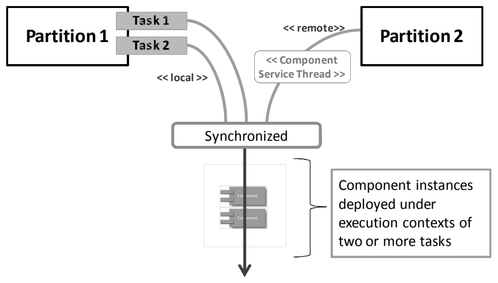

Partition 1
Task 1
Task 2
<< local >>
<< remote>>
<< Component Service Thread >>
Partition 2
Synchronized
Component instances deployed under execution contexts of two or more tasks

Figure 4: Automatic Synchronization.

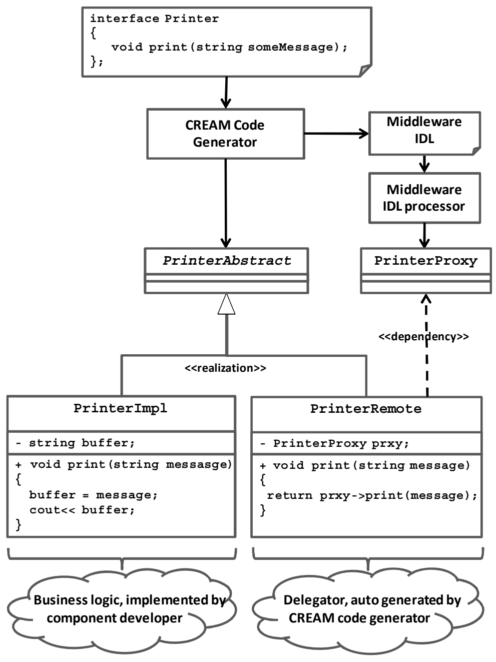
The task model of CREAM enables automatic synchronization among shared component instances. This mechanism is described in Figure 4. The CREAM code generator statically parses the software assembly descriptor (SAD) to analyze for shared component instances used by two or more active components. The code generator then automatically embeds code that uses underlying OS task synchronizing primitives such as a mutex and semaphore to coordinate access to these shared component instances.

The integrated task model of CREAM helps in using external WCET analysis tools within the CREAM component framework. The CREAM component framework, having the complete knowledge of all the tasks in the system, can automatically configure these WCET tools to evaluate the worst case execution time for all tasks.

4.3. Fast Inter-component Communication

The CREAM achieves inter-component communication performance efficiency for collocated inter-component method calls by mapping collocated components' port composition to local function calls and remote components' port composition to communication middleware based remote function calls.

The composition optimization is achieved using polymorphism and the delegator design pattern. The port interface type is associated with an abstract class. This abstract class has two implementations: (1) the actual business logic implementation of interface methods and (2) the delegation implementation to a proxy that handles remote object communication. CREAM's code generator automatically generates the second implementation. The collocated inter-component calls are mapped to the actual business logic implementation method. The remote inter-component calls are mapped to the auto generated delegation implementation method. This entire mechanism is visualized in Figure 5.



```
interface Printer
{
   void print(string someMessage);
};
```

```
- string buffer;
+ void print(string messasge)
{
  buffer = message;
  cout<< buffer;
}
```

```
- PrinterProxy prxy;
+ void print(string message)
{
 return prxy->print(message);
}
```

Figure 5: Interface Methods Local and Remote Implementation.

The build-time binding of component ports to appropriate local or remote references provides optimization and efficiency over that of run-time component frameworks. Those component frameworks usually bind all component ports at run-time and use their communication middleware for inter-component method calls. This communication middleware overhead for collocated inter-component communication is completely avoided in CREAM.

4.4. Transparency from Underlying Communication Middleware

In the CREAM component framework, the component model and operations on components such as component construction, composition and deployment are made independent of the underlying communication middleware. This separation is achieved by developing thin abstraction layer for the communication middleware, having minimal requirements on object-based communication middleware, and code-generation tools. Specifically, it only requires object methods marshalling and un-marshalling support from the communication middleware. Any communication middleware which support this minimal requirement can be used in the CREAM component framework.

The CREAM code generator helps achieve the separation of application logic code from the communication mechanisms. The code generator automatically associates the application business logic object to communication middleware's object servants. It then extracts the business logic object's information from communication middleware's object proxies. This preserves the investment done on developing the actual business logic of components and enables the components to be deployed over different communication middlewares.

5. Implementation and Experimental Results

We have implemented the CREAM framework using the standard C++ programming language and the code-generator in Perl scripting language. We have developed and tested the CREAM on two OS platforms: Linux (2.6.22 kernel) and Windows XP. On Linux, `gcc` (4.1.3) compiler, and `autoconf` (2.61) and `automake` (1.10) build-toolsets were used. On Windows, Visual Studio 2005 was used to develop the CREAM.

We have compared the CREAM performance characteristics to a socket based raw implementation and the MicoCCM. In the raw implementation, method calls between collocated components were handled through local function calls, and method calls between two *partitions* were handled through socket communication. This raw implementation allows us to compare the communication performance for best obtainable values. On the other hand, MicoCCM has been used in many distributed real-time embedded system applications.

We used two computing hosts with the following configuration for our experiments: Intel Centrino 2.80 GHz running Linux 2.6.22 kernel and having 1 GB of RAM memory. The CREAM component framework made use of the Ice-E communication middleware in these experiments.

We measured inter-component communication time for three scenarios. First, the inter-component communication time for components in the same address space was measured. Second, the inter-component communication time for components residing in different address spaces, but within the same host was measured. Third, the inter-component communication for remote components residing in different hosts was measured.

Table 2. Inter-component Communication time

	CREAM	Raw	MicoCCM
Collocated components in the same address space	1.43 µs	1.10 µs	2.74 µs
Remotely located components in the same host	43.4 µs	37.2 µs	76.5 µs
Remotely located components in different hosts	351 µs	332 µs	387 µs

As shown in Table 2, for collocated components in the same address space, the communication overhead of CREAM compared to the raw implementation is 30% whereas MicoCCM causes 149%. For remotely located components in the same and different hosts, the overhead of CREAM is 16.6% and 5.7%, respectively. Compared to this, the overhead of MicoCCM was 106% and 16.6%, respectively.

Table 3. Framework Memory Consumption

CREAM Processes	Size (MB)
DomainExecution-Manager	48
Partition A (on computer 1)	48
Partition B (on computer 2)	40
Total	**128**

MicoCCM Processes	Size (MB)
Naming-service	24
mico-ccmd (daemon on computer 1)	48
component-server (on computer 1)	56
mico-ccmd (daemon on computer 2)	40
component-server (on computer 2)	48
Total	**216**

Table 3 shows that the CREAM component framework makes use of three OS processes: `DomainExecutionManager`, `PartitionA` and `PartitionB` to implement the experimental system of Figure 8 (c) on two hosts. On the other hand, Table 4 shows that MicoCCM uses five OS processes: `Naming-service`, two `mico-ccmd` processes and two `component-server` processes for the same experimental setup. As can be inferred from Tables 3 and 4, CREAM uses 40.7% less memory than MicoCCM. This is achieved through the removal of naming-service and dynamic-composition features of CCM which are rarely needed for an embedded application.

6. Conclusion

In this paper, we have proposed the CREAM as a new generic component framework for distributed embedded systems. We have identified the design requirements of a component framework that meets the challenges of distributed cross-domain applications. We have designed and implemented the CREAM component framework, which is lightweight on memory usage, has integrated task-execution model, efficiently handles inter-component communication, and supports distributed processing in a communication middleware transparent manner. The CREAM component framework was evaluated and compared to a raw socket-based implementation and the MicoCCM.

References

[1] Ivica Crnkovic, and Magnus Larsson, Building Reliable Component-Based Software Systems, Artech House, 2002

[2] C. Szyperski, D. Gruntz, and S. Murer, Component Software: Beyond Object-Oriented Programming, second ed. Addison-Wesley, 2002.

[3] Kung-Kiu Lau, Zheng Wang, Software Component Models, IEEE Transactions on Software Engineering, vol. 33, no. 10, October 2007

[4] Lightweight CORBA Component Model (CCM), OMG Final Adopted Specification, ptc/03-11-03, http://www.omg.org/docs/ptc/03-11-03.pdf

[5] Joint Tactical Radio Systems. Software Communications Architecture Specification V.3.0, August, 2004. http://sca.jpeojtrs.mil/

[6] Common Object Request Broker Architecture: Core Specification, http://www.omg.org/technology/documents corba_spec_catalog.htm , Mar. 2004.

[7] R. van Ommering, F. van der Linden, J. Kramer, and J. Magee, "The Koala Component Model for Consumer Electronics Software," Computer, vol. 33, no. 3, pp. 78-85, Mar. 2000.

[8] AUTOSAR Development Partnership, "AUTOSAR architecture", available at www.autosar.org, Autosar GbR

[9] PECOS Project: http://www.pecos-project.org/

[10] MicoCCM: http://www.fpx.de/MicoCCM

[11] Ice-E: http://www.zeroc.com/icee/index.html