A Component-based Approach for Embedded Software Development

I-Ling Yen, Jayabharath Goluguri, Farokh Bastani, Latifur Khan
University of Texas at Dallas

John Linn
Texas Instruments

Abstract

The rapid growth in the demand of embedded systems and the increased complexity of embedded software pose an urgent need for advanced embedded software development techniques. Software technology is shifting toward semi-automated code generation and integration of systems from components. Component-based development (CBD) techniques can significantly reduce the time and cost for developing software systems. However, there are some difficult problems with the CBD approach. Component identification and retrieval as well as component composition require extensive knowledge of the components. Designers need to go through a steep learning curve in order to effectively compose a system out of available components.

In this paper, we discuss an integrated mechanism for component-based development of embedded software. We develop an On-line Repository for Embedded Software (ORES) to facilitate component management and retrieval. ORES uses an ontology-based approach to facilitate repository browsing and effective search. Based on ORES, we develop the code template approach to facilitate semi-automated component composition. A code template can be instantiated by different sets of components and, thus, offers more flexibility and configurability and better reuse. Another important aspect in embedded software is the nonfunctional requirements and properties. In ORES, we capture nonfunctional properties of components and provide facilities for the analysis of overall system properties.

1 Introduction

Dramatic advances in computer and communication technologies have greatly reduced hardware costs and improved their performance and reliability. This has made it economically feasible to extend the reach of automation to more and more services, such as patient monitoring systems, transportation, communication systems, handheld devices, etc. The market for these embedded computer systems is huge and is expected to grow rapidly over the next few years. For example, 260 million cellular phones were sold in 1999, and this pace will continue with the introduction of new 3G cellular infrastructure. Meanwhile, embedded software continues to become more and more complex due to the growing sophistication and complexity of modern applications. For example, consider telecommunication systems. Just a few years ago, all that a switching system had to do was to establish a route for a call, monitor the call for billing purposes, and release the resources dedicated to the call after it was completed. In recent years, this simple scenario has become extremely complex with an explosive growth in the number of features and capabilities. Telecommunications systems must now handle stationary and mobile calls (both cellular and satellite wireless systems), handle various failure modes (switches, trunk-lines, satellites), support voice and data transmissions, and provide numerous user-oriented features (call forwarding, speed dialing, caller id, etc.). As can be expected, we will face a great challenge in the development of embedded software over the next few years.

Component-based development (CBD) techniques have been widely studied to enhance the productivity of developing complex applications. These techniques can benefit the software development process for embedded systems as well as other application domains. However, CBD approach still faces some difficult problems. First, the developers have to be able to effectively retrieve related components. The retrieval process involves matching the desired functionality and making sure that the component satisfies required properties such as timing requirements and resource constraints. Thus, the developers need to have in-depth knowledge of the available components and their properties. After identifying the components, the developers also face a steep learning curve to master the use of these components.

First, we consider the component retrieval problem. Over the past decade, component retrieval techniques have been studied extensively [MIL94, ZAR97]. Formal methods have been used [MIL94, ZAR97] to achieve better component retrieval. There are two major approaches along this direction. In syntax-based retrieval, component selection is based on matching the signatures of the operations, such as input/output parameter types [LUQ99, ZAR97]. Since it does not provide a complete behavior description, it is not suitable for partially specified retrievals. Semantics-based approach specifies a component by its behavior. Behaviors of components are specified in formal languages [MIL94, ZAR97]. Theorem proving or rule-base reasoning techniques can be used to infer equivalence or similarity of component behaviors [OST92]. These are elegant solutions; however, they require the component developers and users to have extensive knowledge of formal specification techniques and are difficult to use due to the low-level granularity of formal specification.

One issue in component retrieval that has been overlooked frequently is matching of nonfunctional attributes of the components. For embedded systems, nonfunctional attributes, such as time limit, memory constraints, etc., are frequently crucial factors. Thus, component retrieval has to not only consider matching the functional requirements but also satisfy the nonfunctional requirements (NFR). When multiple components can be selected for a given functional specification, the one that best satisfies the NFRs should be selected. Automated tools can be provided to assist with the selection. In [TRA99], an NFR-Assistant tool is presented which assists with the exploration of design alternatives through a graphical interface. In [STE93], the real-time requirement is addressed and tools are provided for selection of components satisfying real-time constraint.

Component retrieval techniques and NFR-based tools can help identify appropriate components for program construction; however, the issue of component composition still remains. The composition process requires extensive knowledge of the involved components. Developers need to obtain knowledge such as calling sequences, component parameterization alternatives, side effects of components, etc. Sometimes, customization or wrapping is required in order to put the component in use for specific environments. Some existing repository systems provide CASE tools to assist with the architecture design, application system modeling, and/or consistency checks [BRO99]. Java library provides implementation stage assistance through code examples. Partial code segments for achieving basic functions are given to illustrate the use of the corresponding components. Java Bean provides a graphical user interface for component parameterization and composition. These are effective approaches in helping with the component assembly process, but due to their ad hoc nature, their applicability is limited.

Along another direction, code synthesis techniques have been used to facilitate the component integration. In Ptolemy [BUC92], a framework is provided for simulating and synthesizing embedded systems. It supports several models for component interaction, including synchronous dataflow, discrete event, etc. User can compose a system by interconnecting the input/output ports of building blocks through a graphical user interface. Each building block is the composition of a set of components and/or other building blocks. Thus, a more complex system can be built hierarchically. Once a system is composed from components, Ptolemy provides tools for code synthesis which mainly deal with buffer management and component scheduling. Several buffer optimization algorithms have been developed to effectively manage the memory space. Component scheduling in Ptolemy is mainly based on various component interaction models and the buffer optimization algorithms.

Chinook is a successor of Ptolemy and is intended for control-dominated systems [BOR95]. It supports design space exploration by the use of a single system specification that captures the reactive real-time behavior of the system and appropriately abstracts interactions with the environment to enhance retargetability. Several interface synthesis techniques are employed to interconnect system components. The necessary interface hardware and software is generated automatically and minimal glue logic is introduced. At the lowest level, Chinook synthesizes device drivers from timing diagrams. Chinook also generates customized processor code and hardware designs.

In this paper, we discuss an integrated approach for embedded software development. We develop an on-line repository for embedded software (ORES) and based on the repository, a set of tools are developed to assist component composition and analysis. To facilitate component-based embedded software development, ORES incorporates several important features. Firstly, we use an ontology-based approach to organize the components in the repository to facilitate effective component retrieval. Ontology is a collection of nodes and their relationships, which collectively provide an abstract view of a certain application domain. The use of ontology repository facilitates convenient repository browsing and provides the handle for more effective search. The design of the repository is discussed in Section 2. Secondly, we use code template to assist component composition. A code template can be instantiated by alternative sets of components and, hence, can be configured and reused for different execution environments and/or user requirements. With code templates, programmers can proceed with component composition without in-depth knowledge of the components. The code template approach and its configurability are discussed in Section 3. Thirdly, based on code templates and component properties, we perform NFR analysis to select appropriate components for instantiation and determine component configurations, such that the requirements of the system are satisfied. To facilitate the NFR analysis, ORES needs to effectively capture nonfunctional behaviors of components. The issues related to NFR properties capturing and analysis are discussed in Section 4. Section 5 states the conclusion of the paper.

2 ORES Component Retrieval

One major design in ORES is the repository system that organizes the components for easy retrieval. We apply information retrieval techniques for the repository design. The components in the repository are organized using an ontology to facilitate easy navigation and effective search. Ontology provides the conceptual view of the application domain and, hence, effectively classifies the components and describes their relationships. However, due to the boundary of packages, software components frequently hold additional relations beyond the semantic relations. For example, in a voice over IP system, the sender needs to *encode* the voice stream and the receiver needs to *decode* the stream. Nodeual-based ontology may capture the relation between different versions of "encoder" or "decoder" functions, but not the "*syntactical*" correlation between a specific pair of "encoder" and "decoder" functions from the same package that have to be invoked in pairs. A consequence of this issue is the necessity of providing additional views in the ontology, for example, one hierarchy is based on the classification of functionality of components and another retains the boundaries of software packages. Due to the potentially complex structure, an ontology with multiple hierarchies is not suitable for browsing. In our approach, we use an **echoing** technique to merge the multiple views into a single hierarchy during browsing while still retaining the characteristics of multiple views.

We first consider a major repository hierarchy that retains software package boundaries. Each node in the repository ontology has a **type**, which can be domain, subdomain, package, abstraction group, abstraction, function group, or function. At the highest level, we have a repository root node. In the hierarchy, software packages are considered as the intermediate units. Under the root, we build the repository ontology by classifying packages according to their functions and application domains. This process is similar to conventional ontology construction. At this level, a node in the ontology represents a **domain** or a **subdomain**. A domain is a major software area, for example, embedded system, operating systems, etc. A subdomain further divides a domain into finer categories. Within a subdomain (or even a domain), there are various software **packages**. In general, a software package consists of a number of **abstractions**, where each abstraction is a program unit that encapsulates certain abstract concepts or behaviors together with some state information that is accessible within the abstraction. A group of **functions** is implemented in an abstraction to access the encapsulated state information and/or achieve certain goals of the abstraction. In a large package, there may be hundreds of abstractions. We use **abstraction groups** to structure abstractions in a package into a hierarchy. Similarly, a large abstraction can have a large number of functions. To allow easy retrieval of the functions, we use **function groups** to structure functions in an abstraction.

Here, we use Java Networking package as an example to illustrate the ontology construction and discuss the echoing scheme in ORES. In Figure 1, we show the original program hierarchy in **java.net**.

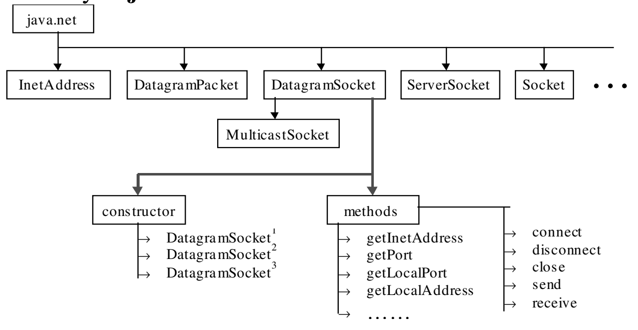

Figure 1. The hierarchy in package java.net.

The java.net package contains several classes. Each class contains a set of constructors and a set of methods. Note that only a part of the package hierarchy is illustrated. In ORES, the classes in java.net are abstractions. We further classify these abstractions into abstraction groups. Here, a sensible way to classify the classes is to have "reliable-communication", "unreliable-communication", and "multicast-communication". Within each class group, we further identify two sub class groups, "packet" and "socket". Under "unreliable-communication.packet", we have the actual class "DatagramPacket" and under "unreliable-communication.socket", we have the actual class "DatagramSocket". Within each class, we group functions into function groups. In the class DatagramSocket, we have function groups "constructors", "get/set-socket-address-information", "channel-establishment-functions", and "message-passing-functions". The ontology in ORES for the java.net package is shown in Figure 2.

Another issue in software repository ontology is that software components may have their own hierarchy due to the inclusion or inheritance. In Figure 2, we can see the inheritance and use relations. In a large repository, a subsystem consisting of components from the repository can also be a component in the repository. In conventional information hierarchy, the actual object can only be the leaf nodes. In software component hierarchy, components can be associated with nodes at any level in the ontology.

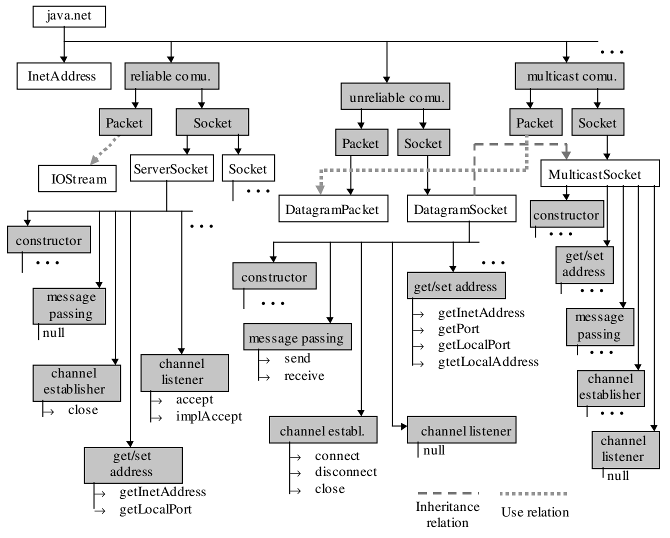

Figure 2. The ontology of java.net package in ORES.

In Figure 2, nodes that belong to echoed hierarchies are shaded. The **echoing** scheme is used to build the same ontology for a group of nodes with similar behavior. Nodes in echoed ontologies are correlated automatically. Echoed ontologies are constructed by taking the "union" of the individual ontologies being echoed. In java.net, the abstractions *ServerSocket*, *DatagramSocket*, and *MulticastSocket* have similar concepts. If we consider component semantics instead of package boundary, then a different hierarchy will be constructed. This hierarchy forms a second view of the ontology. The ontology with multiple hierarchies is presented in Figure 3. As we can see, when multiple hierarchies of the ontology are displayed in a single view, browsing the ontology can be very confusing. Thus, we define a single hierarchy that includes all nodes in the three abstractions. As shown in Figure 2, we add message-passing group in ServerSocket and message-listener group in DatagrameSocket and MulticastSocket abstractions. The added nodes will be empty and their purpose is to relate similar units in a uniform way. So, all socket classes have the same ontology and the nodes in these echoed ontologies are correlated. A pointer "echoed-node" is used to link the root nodes of echoed ontologies together. There is another ontology echoing example in Figure 2. Even though there is no explicit packet classes for multicast-communication and reliable-communication, the node "packet" is still added in each abstraction group.

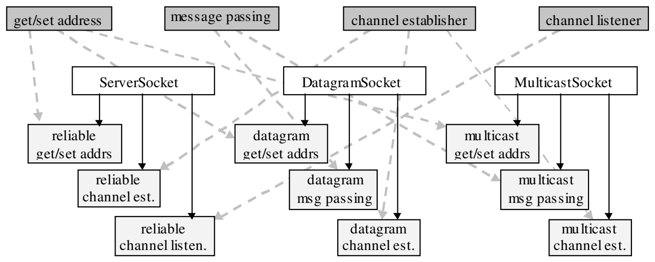

Figure 3. Multiple Views in the Ontology for Java.net.

As we can see, ORES ontology provides a better structure for browsing. With the categorization within the packages and classes, relations among components are better captured. With the echo technique, users can easily understand the relations among various categories and compare them conveniently. Also, once users understand the information in one group, it is easy to extend the knowledge to other echoed groups.

3 Component Composition based on Code Templates

We use code templates as the basis for effective component composition. A **code template** is defined by a set of components and/or code templates (not itself) and the glue code that assembles them together to achieve a certain goal function. We allow overloading for code template definitions. In other words, multiple components can be associated with one code template to implement the specific function defined by the code template. Also, it is possible to map various invocation sequences of multiple components to one template. Thus, a code template is a **configurable unit**. During code synthesis, an appropriate instantiation for a code template can be chosen to best satisfy its nonfunctional requirements and best fit the specific execution environment. The code template definition is shown in the following.

```
template <template name>
    interface
        { in port <port name> [<data size>]; }*
        { out port <port name> <data size>]; }*
        { accept event <event name>; }*
        { raise event <event name>; }*
    end interface;
    body
        { configurable <configurable parameter in main
        body>; }*
        <main body of the template>
    end body
    event
        { on <event name> <actions> end on; }*
    end event
end template
```

The **interface** segment of the template defines the input/output port names and the unit data sizes to be processed or generated. Events that will be processed and may be raised by the template are also defined in the interface. In the code template model, we consider two execution modes, the synchronous data flow and event handling. Under the normal data flow, the main body of the template specified in the **body** segment is activated when it is scheduled to run. When an event is delivered to the code template, the corresponding actions defined in the **event** segment will be activated.

When the code template is first defined, the interface of the code template has to be specified. The body segment can be defined together with the interface or independently. Multiple body segments can be defined for the same template. But the interface should only be defined once. This feature allows alternative components to be associated to the same template to achieve the desired configurability at composition time. Also, event segment in a code template is defined independently with the main body. This allows the events to be handled outside the COTS or third party components if they are not already handled by these components. Also, event handling may be reused even if the components instantiating the code template are different. Default event handling actions have to be defined with the interface definition. Event segment defined together with the body segment overwrites the default event handling definition.

The main body of the code template specifies the composition of components or templates. First, it defines the parameters that are configurable in the main body. Various components may have different configurable parameters. Thus, configurable parameters are always defined within the body segment and associated with the specific main body defined there. There are two possible ways to specify the main body. First, the template can be specified by component invocation statements and glue code in the target programming language like regular coding (defined as follows).

```
{ <regular statements> | <component invocation> }*
```

With this approach, we can map a component to a template without changing its internals. For example, we can have different components implementing various compression algorithms. Without established standards, each component may define a different sequence of parameters. This construct allows the programmer to map the component parameters to the set of unified template I/O ports. Also, this approach allows designer to map various invocation sequences of multiple components to one template. In addition, this approach allows easy specification of glue code that is beyond simple data flow. Otherwise, the glue code, whether can be reused or not, has to be built as a component in order to be composed with the regular components (as the case in Ptolemy). The second type of specification for the template main body is to compose a template from other templates based on synchronous data flow model.

```
source
{ template <template name>; }*
end source
process
{ template <template name>; }
end process
{ connect <connector type> <source port>
<destination port>; }*
```

Let *C* denote the code template to be defined. In the **source** and **process** segments, a set of templates that comprises *C* is specified. The **source** templates are data sources (generators). The **process** templates are processing unit and can only be invoked when its input data are ready. Similar to the Ptolemy, in this template body definition, we specify the connection of the I/O ports of the building blocks (templates) to form the larger building block (or the final task) *C*. However, in our case, each building block is a template that could have multiple instantiations and, hence, are configurable. Note that in/out ports defined in the interface should be connected to some internal ports. For port connection, different types of connectors are allowed. Currently, we define three types of connectors, serial-connector, if-connector, and loop-connector. The serial-connector defines normal data flow. For if-connector and loop-connector, a condition should be specified to make branching and loop control decisions. Definition of specific connector types can clearly specify the external control flow and, hence, facilitates NFR analysis.

Finally, we discuss the specification of event handling actions. Though the events can be substituted by control messages, separating events can give conceptual clarity and allows separate, more efficient mechanism for event delivery. In addition, the event handling functions can be enpowered with different facilities from regular components. More specifically, task control operations such as suspend, continue, stop, and start operations are provided to allow event handler to operate on internal task. Also, event handler can access buffers managed by the system to alter the data flow or insert/delete data units.

4 NFR Analysis for Code Templates

Frequently, there are multiple ways to implement a subsystem from components. To support the construction of configurable code templates can facilitate the construction of adaptive systems. In ORES, we develop tools to evaluate the alternative instantiations for a code template based on NFR requirements and/or execution environment. The analysis can be used to make design decisions. To support this, we need to capture the NFR properties of individual components in the repository and based on individual properties, analyze the cumulative properties of the overall system. To illustrate the concept of component properties in ORES, we start by discussing the general property attributes for components in Subsection 4.1. In Subsection 4.2, we discuss the way ORES collects and stores the NFR properties of individual components. An example of component property attributes collected and stored in ORES is given in Subsection 4.3. In Subsection 4.4, we discuss the basic analysis of system properties based on code templates.

4.1 Attributes for Components Description

Many information attributes are required to describe a component and different components may require different attributes to explicitly describe their characteristics. Thus, we use an adaptive approach to dynamically define the attributes for each component. In ORES, DTD (document type definition) is used for information attributes definition and XML is used to specify the actual measurement data for each component. At the repository root node, we define a basic set of information attributes. These attributes are inherited by all nodes in the repository. A child node inherits the information attributes defined by its parent node and it can modify the definition by adding additional attributes or removing some of them. The information attributes are also node-type dependent. A general specification of the node information is given in the following:

General Information
- node id, node type
- keywords list (with +/– weight)
- short description of the node

Information Pointers
- file name
- pointer type
- code pointers
    - source code: can be file name + range of line numbers
    - executable: file name (many will have the same name)
- document pointers
    - user manual, programmer's guide, tutorials, component description, etc.
    - problem/bug report, reliability information
    - review information, limitations, usage experience report
    - test data, correct output for the test data, test results, operational profiles
- web links

Node Specification (this is node-type specific)
- (These are described in detail later in this report.)

Node Properties (this is mainly used for function nodes and it is domain-specific
- reliability
- portability
- time
- resource requirements
- output quality

Component information is classified into four categories, the general information, information pointers, node specification, and properties. Every node has the "general information" fields, including node-id, node-type, keywords, and short description. Node-id is defined by the path name of the node, which concatenates the names from the root node all the way to the current node in the main hierarchy of the ontology. Node-type has been defined in the previous subsection. A node can be of type "domain", "package", "function", etc. The "keywords" field is maintained to facilitate search. It modifies the set of keywords and their weight derived from the documents associated with the node.

The information pointers are a set of pointers referencing external information files. For software components, we maintain pointers to the source code and/or executables. Other pointers are required for various documents and web links. Components will have very different sets of pointers, but all pointers are handled uniformly. Each pointer is specified by a file name and the pointer type. There is no need for a node to redefine specific information pointer categories when it contains different pointers from its parent. Pointer type is useful to understand how to use the specific information in the file. Currently, we consider several major pointer types, including source code pointer, executable pointer, component description document pointer, review document pointer, test data pointer, and operational profile pointer. Further classification and type definitions will be given when necessary.

Specification attributes describe the characteristics of a node and it is node-type dependent. The specification for each node type is given as follows.

- function
  - function specification
    - function name, input/output parameters, exceptions
  - function type: constructor / data manipulation functions (like get-set) / data abstraction functions (like add-delete-update) / conventional functions (e.g., sort, matrix-operations, etc.) / IO functions / database functions
- function group
  - description
- abstraction
  - abstraction specification
  - global data, global definitions, etc.
- abstraction group
  - description
- package
  - version number, description and comparison of other versions
  - license information, prices, vendor, general system requirements
  - package-wide definitions: exceptions, glossaries
- subdomain or domain
  - description

For function nodes, the API specification (input, output, exceptions) should be provided. For abstraction nodes, the abstraction API, such as interface functions and exceptions, and inheritable or externally accessible variables should be provided. For a package node, the package version number, new release information, licensing information, price information, vendor, etc. should be provided. The execution environment requirements, such as OS, processor speed, memory and disk size, devices, etc., should also be specified. Some package-wide glossary definitions should also be provided. For other node types, there is no formal specification and overall description of the node is sufficient.

The property information refers to the attributes of a component that can be measured with formally defined metrics. It is highly domain-dependent. The list of property attributes for a domain/subdomain can be defined at a domain/subdomain level node and inherited by its descendant nodes of specific types. We classify the property information into five categories, including time property, resource requirements, output quality, reliability, and portability. Generally, time property, resource requirements, and output quality are interdependent. The input domain also impacts these measures. For embedded software, time, resource requirements, and output quality are important properties and we will discuss them in the next section. Reliability and portability will not be discussed in this document.

4.2 NFR Property Information Attributes

In ORES, we consider three major categories of component attributes, namely, time, resource requirements, and output quality for embedded systems. The list of measurable attributes in each category is given in the following.

- Time measurements (average and worst case values)
  - CPU cycles
  - I/O time
  - communication latency
  - execution time (in terms of real-time)
- Resource requirements (average and worst case values)
  - memory requirements
    source: by program / data
    persistent (stays after completing the execution of the component)
    / volatile (returned to the system after finishing execution)
    static (fixed amount of memory allocated at the beginning of the execution)
    / dynamic (variant amount memory allocated at run time)
  - disk requirements
  - power consumption
  - communication channel capacity requirements
- Output quality (highly component dependent)

Note that memory requirement can come from program or data. Also, memory required for data can be persistent or volatile and can be allocated statically or dynamically. Output quality attributes can vary greatly depending on the nature of the components. It is necessary to analyze the target component in order to identify all the desired output quality attributes.

In general, a component itself does not independently determine these property attributes. First, the execution platform will greatly impact these measures. For example, the

CPU speed, memory type (such as on-chip/off-chip cache, RAM), and OS can greatly impact the execution time. Second, for some components, the input domain can have significant impact on these measures. For example, an input voice stream with high sample rate will require longer processing time and larger memory size. Finally, time, quality, and resource requirements can impact each other through adjustable parameters within the component. For example, we may be able to reduce the memory requirement of a program by reducing some buffer size, which may, in turn, impact the execution time and output quality. Or, we can alter an algorithm to perform certain computations that may improve the execution time but degrade the output quality. These property trade-off information can be very helpful for designing adaptive systems that can be easily ported to different platforms with different resource constraints or adapt to different execution conditions.

In order to correctly measure the component property attributes, we need to identify all parameters that can impact these measurements. As discussed above, we need to identify (i) input parameters, (ii) execution environment parameters, and (iii) parameters in the component that are configurable. Besides the execution environment parameters, the other two types of parameters are highly component dependent. We first list some major environment parameters.

- CPU
  - speed
- memory
  - on-chip, off-chip cache, RAM
  - memory/bus speed
- I/O devices (disk, network, etc.)
  - latency
  - transfer rate
- OS

Before we proceed to the identification of input and configurable program parameters, we need to identify all the attributes we wish to measure for the component. For output quality, we need to analyze the component in order to determine the quality attributes and define the metrics for measuring the quality. Let M denote the set of measurable attributes identified for a component C. Next, we should determine the parameters that can impact our measurements. Input parameters can be determined by analyzing input parameters of the component and their related properties. By looking through the attributes in M, we can identify the possible parameters in the program that can impact the measurement of one or more attributes in M. Let I denote the set of input parameters, E the set of environment parameters, and A the set of configurable parameters that have been identified. Now, I, E, and A form the n-dimensional parameter space (where n = the total number of parameters in I, E, and A) and each attribute in M can be measured and plotted against the parameter space, i.e., we can obtain $M_i = f_i (I, E, A)$.

In some cases, a simple f_i, where $M_i = f_i (I, E, A)$, can be derived in a straightforward way without any measurements. Also, in some rare situations, f_i may be derived easily from the measured data. In many cases, it may be difficult to obtain the functional representation f_i for some attribute M_i (due to the limited measurements we may be able to obtain). Thus, we allow two forms of expressing the mapping from (I, E, A) to M_i. Either the functional representation f_i or the raw measurement data can be stored in the repository. In the case of the functional representation, we confine it to be polynomial functions and the coefficients are maintained. In the case of raw data, we store the values of the parameters and the corresponding measurements for the attributes in the repository. More specifically, the values for parameters $(I_1,I_2, \ldots, E_1,E_2, \ldots, A_1,A_2, \ldots)$ and the data measured for $(M_1,M_2, \ldots)$ are stored in the repository. With sufficient amount of measurements, we can use interpolation techniques to predict the data for the entire parameter space. Otherwise, the available data will be used for various analysis without interpolation.

4.3 NFR Properties for an AEC Component

As an example, we consider an Acoustic Echo Canceller (AEC) component provided by Texas Instruments for TMS320C55x. The component uses adaptive filtering with 2nd order affine projection algorithm [OZE84].

Measurable Attributes

Time measurement: CPU cycles.

Resource requirements: memory requirements by the program and the data.

Output quality: The main quality parameter for an AEC is the Terminal Coupling Loss (TCL).

Parameters that can Impact the Measurable Attributes

Input parameters. The input parameters for the AEC include

- sampling rate
- echo trail length
- AEC Filter length = sampling rate * echo trail length

Configurable program parameters.

- Frame size

The echo trail length parameter is determined based on the environment that will provide the AEC with its inputs. For example, the environment could be a conference room, car environment, auditorium, etc. Higher sampling rates and echo trail length demand more filter length and memory and, hence, more processing time. The filter length affects the length of the trail of echo that can be canceled by the particular configuration. For example, a filter length of 256 can be used to cancel echo trails up to 32 ms (at sampling rate of 8kHz) and up to 128ms for a filter length of 1024 (at 8kHz). The decision of which filter length to choose depends on the scenario in which the AEC will be used, i.e., the characteristics of the room. In the case of a hands-free car environment, filter length of 256 (i.e., 32 ms echo trail) is sufficient. But if the component is used in video conferencing application (usually used in a conference rooms), a filter length of 1024 (i.e., 128ms echo trail) will be more appropriate. Hence, this parameter must be implemented as a configurable parameter so that the application designer can customize it according to his needs. Frame size is a parameter that effects the rate at which the AEC is called. Having a smaller frame size could result in more overhead due to the higher frequency of function calls.

From experimental study, we collected nonfunctional behavior for the AEC component which is shown in the following six diagrams. Figures 1 and 2 show the impact of filter length and frame size to the average execution time. Figures 3 and 4 show the memory requirements with the same alternations in filter length and frame size parameters. With the same alternations, the quality of the output are affected and the impact is shown in Figures 5 and 6.

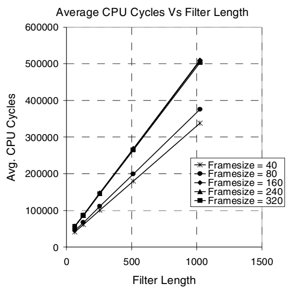

Figure 1

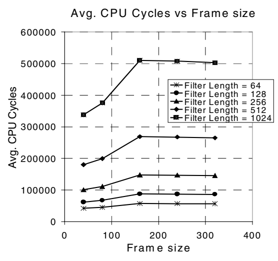

Figure 2

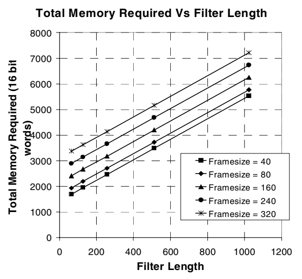

Figure 3

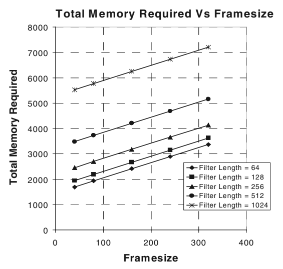

Figure 4

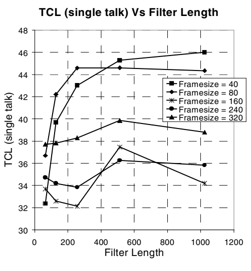

Figure 5

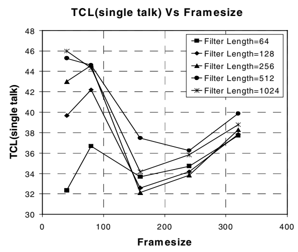

Figure 6

As we can see, the data from Figures 1, 3, and 4 are linear and can be stored in the repository as a polynomial. The data shown on Figures 2, 5, and 6 are stored in a two dimensional matrix with each dimension defined by one of the configurable parameters, filter length and frame size. The nonfunctional property data stored in ORES for the AEC component is shown in the following.

Input parameters
    sampling rate = 8 KHz
    echo trail length = 32msec
Environment parameters
    board = OMAP 1510 EVM
    OS = DSP/BIOS II and Symbian's EPOC
Configurable program parameters
    filter length = [64, 128, 256, 512, 1024]
    frame size = [40, 80, 160, 240, 320]
Measurable Attributes
    average CPU cycle
    memory requirement
    quality
Measurement
    f (*memory requirement*) = *filter length*, *frame size*, (4.0, 6.0, 1192.74)
    A (*average CPU cycle*) = *filter length*, *frame size*, (338000, 375212, 510304, 507858, 502782, 179915, 199205, ……)
    A (*quality*) = *filter length*, *frame size* (32.36, 36.69, 33.67, 34.71, 37.71, 39.66, 42.20, ….)

All the parameters (italic font in the above specification) are defined by a DTD at a domain or subdomain node in the ontology for a specific application domain. For a parameter with a fixed value (such as input parameters and environment parameters in the above example), the value is given. For a variant parameter, (such as the configurable program parameters in the above example), "[]" is used to specify the various data selected for the measurement. Here we have selected filter length = 64, 128, 256, 512, and 1024 and frame size = 40, 80, 160, 240, and 320. In the measurement section, the actual data for measurable attributes are specified, either by a function or by an array which stores multiple dimensional data. For example, the memory requirement attribute data is a linear function of filter length and frame size and, hence, is specified as 4.0 * *filter length* + 6.0 * *frame size* + 1192.74. Note that, even though average CPU cycle has a linear relation with filter length, we still express the CPU cycle data in an array. As we can see, to store the CPU cycle data for various frame sizes (which is not suitable to be expressed by a polynomial), we anyway have to store the data for various filter lengths. Storing the polynomial function relative to filter length will be redundant. In the array, the data is stored by first iterating through the parameter at the least significant position listed after the = sign. For CPU cycle, the least significant position has parameter frame size. So, the second element in the array is for filter length = 64 and frame size = 80 and the sixth element is for filter length = 128 and frame size = 40.

4.4 NFR Analysis based on Code Templates

Based on the code templates discussed in Section 3, we can model the NFR analysis easily. Consider a code template C, containing n sub-templates c_1, c_2, …, c_n. On the other hand, a set of NFRs, namely, $\{p_1, p_2, …, p_m\}$, can be specified for C (for example, we can have p_1 to be the execution time, p_2 to be the memory requirement, etc.). As we can see, the summary effect of all the components instantiating c_i, for all j, together with the glue code determines the cumulative properties of C. Thus, we should have

$$\sigma_j\,(V_C^{\,j}, V_1^{\,j}, V_2^{\,j}, \ldots, V_n^{\,j})\ \Delta_j\ V_{nfr}^{\,j}\ , \text{ for all } j.$$

Here $V_i^{\,j}$ is the value of property attribute p_j for the component that instantiates c_i; $V_C^{\,j}$ is the value of p_j for the glue code itself; $V_{nfr}^{\,j}$ is the specified value of the NFR requirement for C in terms of property p_j; σ_j is the function for computing the summary effect of p_j; and Δ_j is the comparison operator given in the NFR.

For different types of property attributes, the summary function σ_j and the comparison operator Δ_j maybe different. Also σ_j can be a complicated function since the glue code can contain branch and loop constructs. Here, we consider some example cases. Consider three attributes, with p_1 being the execution time, p_2 being the persistent memory required, and p_3 being the scratch (temporary) memory required. Assume that C only contain sequential statements, no loop or branch constructs. In this case, we have

$$\sigma_1\,(V_C^{\,1}, V_1^{\,1}, V_2^{\,1}, \ldots, V_n^{\,1}) = \Sigma_{i=1..n}\, V_i^{\,1} + V_C^{\,1} \le V_{nfr}^{\,1},$$
$$\sigma_2\,(V_C^{\,2}, V_1^{\,2}, V_2^{\,2}, \ldots, V_n^{\,2}) = \Sigma_{i=1..n}\, V_i^{\,2} + V_C^{\,2} \le V_{nfr}^{\,2}, \text{ and}$$
$$\sigma_3\,(V_C^{\,3}, V_1^{\,3}, V_2^{\,3}, \ldots, V_n^{\,3}) = \mathrm{Max}_i\, V_i^{\,3} + V_C^{\,3} \le V_{nfr}^{\,3}.$$

Frequently, some NFR constraints are not rigid; e.g., instead of a fixed deadline, a range of deadlines may be acceptable. For example, in a soft real-time system, real-time requirements can be expressed by an acceptability function [YEN97, ZAR97]. In this case, specification for Δ_1 should be changed.

$$\sigma_1\,(V_C^{\,1}, V_1^{\,1}, V_2^{\,1}, \ldots, V_n^{\,1}) = \Sigma_i\, V_i^{\,1} + V_C^{\,1} = x$$
$$Q^1\,(\,\sigma_1(V_T^{\,1}, V_1^{\,1}, V_2^{\,1}, \ldots, V_n^{\,1})\,) = \begin{cases} 0, \text{ if } x \ge {}_{high}V_{nfr}^{\,1}, \\ 1, \text{ if } x \le {}_{low}V_{nfr}^{\,1}, \\ A_{nfr}^{\,1}\,(x), \text{ otherwise} \end{cases}$$

Here, ${}_{high}V_{nfr}^{\,1}$ is the upper bound and ${}_{low}V_{nfr}^{\,1}$ is the lower bound for the deadline and $A_{nfr}^{\,1}$ is the acceptability function in terms of p_1. The goal is to optimize Q^1 and satisfy the other two requirements.

5 Conclusion

We have presented an integrated mechanism to facilitate efficient and cost-effective embedded software development. Currently, we have developed an ontology-based component repository built on top of Oracle. It provides effective components retrieval and facilitates capture of component properties. Out next step is to populate the repository with embedded software components and related code templates. We are also developing a framework for code template based component composition. The framework interprets code template definition and manages code template instantiation based on NFR analysis results. It also generates glue code for buffer allocation and management and event delivery. For NFR analysis, the framework is designed to go through the nested templates recursively to compute the cumulative properties. In the first version of the framework, the user specifies specific component instantiations and component configuration parameter values to obtain the analysis results. Algorithms and tools will be developed to determine optimal instantiation and configuration.

6 Acknowledgement

This research was supported in part by the National Science Foundation under Grant No. EIA-0103709, by the Texas ATP under Grant No. 009741-0143-1999, and by Alcatel USA and Texas Instruments, Inc.

Bibliography

[BOR95] G Borriello, P Chou and R Ortega, "Embedded System Co-Design Towards Portability and Rapid Integration" *Hardware/Software Co-Design*, M.G. Sami and G. De Micheli, EDs., Kluwer Academic Publishers, 1995. pp. 243--264.

[BRO99] A.W. Brown, B. Barn, "Enterprise-scale CBD: Building complex computer systems from components," *Proc. Software Technology and Engineering Practice*, Sep 1999, pp. 82-91.

[BUC92] Joseph Buck, Soonhoi Ha, E A Lee and D Messerschmitt "Ptolemy: A Framework for Simulating and Prototyping Heterogeneous Systems" *International Journal of Computer Simulation*, August 1992

[LUQ99] Luqi and J. Guo, "Toward automated retrieval for a software component respository," *Proc. IEEE Conf. And Workshop on Engineering of Computer-Based Systems*, March 1999.

[MIL94] A. Mili, R. Mili, and R. Mittermeir, "Sotring and retrieving software components: A refinement based system," *Proc. Intl. Conf. Software Engineering*, May 1994, pp. 91-100.

[OST92] E. Ostertag, J. Hendler, R. Prieto-Diaz, and C. Braun, "Computing Similarity in a Reuse Library System: an AI-based Approach," *ACM Transactions on Software Engineering and Methodology*, vol. 1, no. 3, July 1992, pp. 205-228.

[OZE84] K. Ozeka and T. Umeda, "An adaptive filtering algorithm using an orthogonal projection to an affine subspace and its properties," *Electronics and Communications in Japan*, Vol. 67-A, No-5, 1984

[STE93] R.A. Steigerwald, "Reusable component retrieval for real-time applications," *Proc. IEEE Workshop on Real-Time Applications*, May 1993, pp. 118-120.

[TRA99] Q. Tran and L. Chung, "NFR-Assistant: Tool support for achieving quality," *IEEE Symp. Application-Specific Systems and Software Engineering and Technology*, Texas, March 1999, pp. 284-289.

[Yen97] I-L. Yen and I.-R. Chen, "Reliability assessment of multiple-agent cooperating systems," *IEEE Trans. Reliability*, Sep. 1997.

[ZAR97] Moormann-Zaremski and J.M. Wing, "Specification matching of software components," *ACM Trans. Software Engineering and Methodology*, Vol. 6, No. 4, 1997, pp. 333-369.