Achieving Component Reconfigurability for QoS Trade-offs: A Rule-based Approach

Jia Zhou, Kendra Cooper, Hui Ma, I-Ling Yen
Department of Computer Science
University of Texas at Dallas
{jxz023100, kcooper, hxm012600, ilyen}@utdallas.edu

Abstract

Component-based development (CBD) is a promising approach for developing high quality software in a cost effective and timely manner. CBD techniques can be applied to real-time, embedded systems. Traditionally, most CBD techniques mainly consider matching functional requirements. However, real-time, embedded systems generally have constraints on footprint, power consumption, real-time, output precisions, etc. Thus, CBD techniques for real-time, embedded systems must consider how to satisfy the QoS requirements. Additionally, QoS requirements often conflict with each other and system designers generally need to consider the trade-offs among them. As a result, it is desirable that components are reconfigurable to facilitate the system QoS trade-off considerations.

Unfortunately, most components are not designed to be reconfigurable in terms of QoS trade-offs. Thus, making the components reconfigurable in terms of QoS trade-offs is an important step towards developing component-based real-time, embedded software.

This paper presents a new component parameterization technique which can transform individual components to reconfigurable ones such that they can support various QoS trade-offs to achieve overall system QoS requirements. The technique has a defined process, is rule based, and is semi-automated. The prototype tool set for the novel technique has been developed and applied to several example components.

***Index Terms*—component parameterization technique, embedded system, QoS trade-offs, rule base**

1. INTRODUCTION

Intensive market competition has been driving software companies to explore fast and low cost software development methods. Component-based development (CBD) has been proposed as an effective approach to decrease the development time and cost by building applications from reusable, composable and exchangeable building blocks: the components [1][2]. CBD techniques can be applied to real-time, embedded systems. Traditionally, most CBD techniques mainly consider matching functional requirements. However, real-time, embedded systems generally have constraints such as footprint, power consumption, real-time, output precisions, etc. Thus, CBD techniques for real-time, embedded systems must consider how to achieve the QoS requirements. Additionally, QoS requirements often conflict with each other and system designers generally need to consider the trade-offs among them. As a result, it is desirable that components are reconfigurable to facilitate the system QoS trade-off considerations.

The advantages of having reconfigurable components in terms of QoS trade-offs are clear in two situations. At design time, system designers can first select reconfigurable components which satisfy the functional requirements and then configure them to satisfy the system's overall QoS requirements. At execution time, adaptive systems composed from reconfigurable components can be conveniently customized to adapt to dynamically changing environments.

Unfortunately, most components are not designed to be reconfigurable in terms of QoS trade-offs. Thus, making the components reconfigurable in terms of QoS trade-offs is an important step towards developing component-based real-time, embedded software. Currently, there are some techniques in the real-time system field which provide effective QoS trade-off support. Even though these techniques do not make use of reconfigurable components to achieve QoS trade-off reconfiguration, they are still deserved investigation since the reconfigurable component can also be used in these techniques to get better results. One approach is scheduling techniques, which are mainly used in operating systems (OS) [3] and multimedia systems [4]. Scheduling techniques permit OS or multimedia systems to schedule higher priority tasks to limited resources so that the QoS requirements of most tasks are guaranteed in a changing environment. However, scheduling

techniques mostly consider satisfying real-time constraints at the task level. The QoS constraints of higher priority task are guaranteed at the cost of sacrificing lower priority tasks. If a task does not have enough resource, then it may not proceed until other tasks release resources. The tasks are not able to adapt themselves to keep working in a changing environment. Middleware approaches, in contrast, can support QoS trade-off adaptation among applications or components [5][6][7][8]. Here, the QoS trade-off analysis is hidden within the middleware so that application developers do not need to worry about QoS concerns [9]. Recently, due to the success of aspect-oriented software development (AOSD) techniques in software design, development, and in particular the success of AOSD techniques in handling non-functional issues in a system, middleware designers have introduced AOSD techniques into middleware design and applied them to handle QoS related issues in middleware systems [10][11][12][13]. Their experiences show that AOSD techniques can further enhance the flexibility and extensibility of middleware in handling QoS trade-off concerns. Middleware approaches can effectively handle QoS trade-offs and support the design of systems that best satisfy the QoS requirements. However, they treat components as black boxes and do not consider reconfiguring them to achieve further QoS reconfigurability.

Scheduling and middleware are effective techniques to support QoS trade-off considerations. However, these techniques do not sufficiently examine components to utilize their own QoS trade-off reconfiguration potential. Component customization techniques, in contrast, are designed to support component adaptation. For example, active interface technique [14] provides a mechanism for filtering method invocation by extending the responsibility of the interface. Binary component adaptation (BCA) technique [15][16] allows components to be adapted and evolved in binary form and during program loading. Superimposition technique [17] enables software engineer to impose predefined, but configurable, types of functionality on a component. Wrapper techniques [18] introduce new behavior that is executed before, after, or in place of an existing method. However, because these techniques are designed to adapt a component's functional behavior, none of them provides a systematic way to change QoS aspects of the component. Although some techniques such as copy-paste [17][19] can be used to customize component's QoS trade-offs manually, they are error

prone, time consuming, and require experts. Hence, a systematic and high impact alternative is desired to make individual component reconfigurable to support different QoS trade-offs.

To complement and extend existing techniques, we investigate fundamental issues in transforming individual components to reconfigurable ones which can support various QoS trade-offs to achieve overall system QoS requirements. Some of the issues investigated include: 1) what features a parameter should have which can affect the component's QoS trade-offs; 2) how to quickly identify parameters with such features in a component; 3) determine if a component can be configured for various QoS trade-offs by changing some parameters' (variable or constant defined in the component) values. A new component customization technique is then proposed which can identify parameters in a component and modify them to adapt the component's QoS trade-offs [20][21]. The features of these parameters are described as predicates in the technique and are organized in a rule base. The rule-based approach enhances the flexibility and extensibility of the proposed component customization technique since rules can be easily added, removed, and modified. A tool set is also developed to partially automate this component customization technique.

The remainder of this paper is organized as follows. Section 2 introduces the basic concepts of the component parameterization technique and illustrates three examples. The complete component parameterization process is presented in Section 3. Section 4 gives a detailed discussion of configurable parameter identification and rule base. The design and implementation of the tool set is described in Section 5 followed by an introduction of the application of the technique in Section 6. Section 7 presents conclusions and future work.

2. BACKGROUND AND EXAMPLES

The development of a real-time, embedded system with multiple QoS requirements can be greatly facilitated if the QoS properties of software components are reconfigurable. Many real world cases demonstrate that different QoS trade-offs can be accomplished by adjusting certain parameters in the program. For example, consider the Newton's algorithm which is a numerical method for finding the roots of a polynomial. Given a polynomial $f(x)$, Newton's algorithm starts with an initial guess x_0 for

root x, then repeatedly refines the approximation by computing equation $x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$ until $|x_{n+1} - x_n| < \varepsilon$. As can be seen, the ε value affects the Time-quality trade-off. With a larger ε value, the program generates a root with less accuracy but the computation takes less time. With a smaller ε value, the effect is reversed. This is a very common type of parameter that can be adjusted to balance the trade-off between time and output quality.

The goal of our work is to develop an accurate, effective approach for identifying such parameters in software components and use the identification result to customize the components into reconfigurable ones in term of QoS properties. The details of the approach are discussed in subsequent sections. This section introduces some of the concepts and terminologies used throughout the paper and examples illustrating various QoS trade-offs.

2.1. Basic Concepts

The fundamental concepts of a component, QoS properties, configurable parameter, and QoS properties reconfigurable component used in this paper are presented here.

Component. A component refers to a module that contains both code and data. It presents an interface that can be invoked by other components.

QoS properties of a component. Major categories of QoS properties include time, resource requirements, output quality, battery consumption, etc [22]. Examples of time properties include CPU cycles, I/O time, communication latency, and execution time (in terms of real time). The measurements on these time properties may include average, worst case, or standard deviation values. Examples of resource requirements properties include memory requirements, disk requirements, power consumption, and communication channel capacity requirements. The memory requirement can be for program or data. Also, the memory required can be persistent (the data stays after completing the execution of the component), volatile (returned to the system after finishing execution), allocated statically (fixed amount of memory allocated at the beginning of the execution), or allocated dynamically (variant amount memory allocated at run time). Examples of output quality properties include precision and accuracy. Output quality properties 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 properties.

Configurable parameter. A configurable parameter is a constant or a variable defined in a component which, when adjusted, can provide QoS trade-offs. The formal definition of the configurable parameters is as follows:

Let $M = \{M_i \mid \text{for all } i\}$ denote the set of QoS properties to be considered.

Consider a component C. Let $CP = \{p_j \mid \text{for all } j\}$ denote the set of configurable parameters in C.

Now:

1. After changing the setting of p_j, the functional requirement of C is still satisfied. The setting of p_j can be the type and initial value of p_j. If p_j is an array variable, then the setting can be the size of the array.
2. By changing the setting of p_j, at least one property M_i, for some i, changes significantly.

Trade-offs. Trade-offs can be achieved by adjusting configurable parameters. Common types include time-space, time-quality, and space-quality trade-offs. Time-space trade-offs may be found in function calls that use static values calculated in a previous, possibly different function call. If the amount of memory needs to be reduced, then the value could be recalculated, using additional CPU cycles, rather than stored. Space-quality trade-offs may be found in a statically allocated array (e.g., a buffer). If the amount of memory needs to be reduced, then the size of the array could be reduced at the cost of decreasing the quality of the output. Time-quality trade-offs may be found in loop control parameters. If the performance needs to be improved, then the termination condition may be changed such that the loop terminates earlier.

QoS properties reconfigurable (QoS-R) component. A QoS-R component is a software component together with a delta file. The QoS-R component allows the software component to be reconfigured either statically or dynamically to provide desired QoS trade-offs. The delta file records a group of mappings between the modification of the software component and corresponding QoS property measurements of the component when applying the modification. The data measurements for original component are also records in the delta file. However, they are mapped to empty modification. Given the QoS requirements of a component at design time, the delta file of the component is checked

to determine the QoS property measurements which best approximate the QoS requirements. The reconfiguration is then accomplished by applying the modifications corresponding to the QoS property measurements.

2.2. Examples

This subsection presents three examples to illustrate three QoS trade-offs including time-quality, time-space, and space-quality trade-offs. These examples are also used in Section 6 to demonstrate the application of our technique.

2.2.1 Example for Time-Quality Trade-off: Newton's algorithm

The program code for Newton's algorithm, described at the beginning of this Section, is listed in TABLE I. Note that *f* (in line 4) is the input function for which the root is computed and *df* (in line 4) is the differentiation of *f*.

TABLE I. NEWTON'S ALGORITHM BEFORE CUSTOMIZATION

```
1   #include ...
2   #define epsilon 0.001
3
4   float Newton (float (*f)(float), float (*df)(float), float start)
5   {     float xold, xnew;
6         int num;
7
8         xold = start;
9         xnew = xold;
10        num = 0;
11        do
12        {     xold = xnew;
13              if ( df(xold) == 0 )
14                  return (divide_by_zero);
15              xnew = xold – (f(xold)) / (df(xold));
16              num++;
17        }
18        while ( fabs(xold–xnew) > epsilon );
19        return (xnew);
20  }
```

2.2.2 Example for Time-Space Trade-off: Random number generator

The algorithm of random number generator is implemented in two steps. First, the user initializes the seed values. Second, every time the user needs a random number, the random number generator computes the new random number from current seed values. After the new random number is generated, the generator updates seed values from the current seed values. The program code of the generator is listed in TABLE II. The example code generates random numbers uniformly distribute from 0 to 2^{32}–1. Initialization is done by calling the function *sgenrand*(*seed*) with an unsigned long

integer seed. The seed value should not be ZERO; different seeds give different sequences.

TABLE II. RANDOM NUMBER GENERATOR BEFORE CUSTOMIZATION

```
1     #include <stdio.h>
2     /* Period parameters */
3     #define N 624
…     …
15    static unsigned long mt[N]; /* the array for the state vector  */
16    static int mti=N+1; /* mti==N+1 means mt[N] is not initialized */
17
18    /* initializing the array with a NONZERO seed */
19    void sgenrand(unsigned long seed)
20    {    /* setting initial seeds to mt[N] */
21    mt[0] = seed & 0xffffffff;
22    for (mti=1; mti<N; mti++)
23    mt[mti] = (69069 * mt[mti–1]) & 0xffffffff;
24    }
25
26    unsigned long genrand()
27    {    unsigned long y;
28    static unsigned long mag01[2]={0x0, MATRIX_A};
29    /* mag01[x] = x * MATRIX_A  for x=0,1 */
30
31    if (mti >= N) { /* generate N words at one time */
32    int kk;
33    if (mti == N+1)   /* if sgenrand() has not been called, */
34              sgenrand(4357); /* a default initial seed is used   */
…             generate new values of mt[0..N–1]
46    }
47
48    y = mt[mti++];
49    y ^= TEMPERING_SHIFT_U(y);
50    y ^= TEMPERING_SHIFT_S(y) & TEMPERING_MASK_B;
51    y ^= TEMPERING_SHIFT_T(y) & TEMPERING_MASK_C;
52    y ^= TEMPERING_SHIFT_L(y);
53    return y;
54    }
```

Current seed values are stored in a static array *mt*[*N*] (in line 15) in this program. Each time function *genrand*() is called, the random number is computed from the values store in array *mt*[*N*]. For every *N* calls, the values stored in array *mt*[*N*] are updated once to reflect the current seed values. Differently, a local array *mt*[*N*] can be used to store current seed values so that the time-space trade-off can be adjusted. This strategy requires the program to store the initial seed and compute the values of the array *mt*[*N*] first before computing the random number from these values. The array *mt*[*N*] in this example is another kind of parameter which can impact time-space trade-off. Static array is allocated at compile time and exists throughout program execution in C language. When using a static array to store the current seed values, the program avoids recomputing the current seed values from the initial seed values every time before a new random number needs to be generated and hence can generate the random number with less time. When using a local array to store the current seed values, the program avoids occupying the memory allocated for the array during the entire execution time of the program and hence can save the memory space. However, since the program needs to recompute

the current seed values from the initial seed values every time before a new random number needs to be generated, the code after customization uses more time to generates the random number.

2.2.3 Example for Space-Quality Trade-off: Solving the heat equation

Consider the simple iteration algorithm to compute the approximate solution of the heat equation $\frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x^2}$ with the boundary conditions $u(t,0) = u(t,1) = 0$ and initial conditions

$u(0,x) = 2x, (0 \le x \le \frac{1}{2})$ *and* $u(0,x) = 2(1-x), (\frac{1}{2} \le x \le 1)$.

The iteration equation takes the form: $u(t+1,x) = (1-2r)u(t,x) + r(u(t,x+1) + u(t,x-1))$ where $u(t,x)$ is a heat value at the point x and in time t, $r = \frac{dt}{dx}$, dt and dx are step size in x and t dimension respectively. The condition that $r \le \frac{1}{2}$ is called the stability condition and, if not satisfied can cause the solution to diverge. Finding every subsequent time is simply a case of iterating through the equation finding each value from the three corresponding values in the previous time. The code for the program demonstrating the simple iteration scheme is in the TABLE III. The code demonstrates the heat equation calculated using the simple iteration method for $0 \le x \le 1$ and $0 \le t \le 0.3$.

TABLE III. SIMPLE ITERATION SOLUTION OF HEAT EQUATION BEFORE CUSTOMIZATION

```
1   #include <stdio.h>
2   #include <math.h>
3
4   #define r     0.5
5   #define dx   0.05
6   #define dt    dx*dx*r
7
8   #define X0  1
9   #define XN   (X0 / dx)
10  #define T0  0.3
11  #define TN   (T0 / dt)
12
13  void simpleiteration(void)          /* Numerical solution of heat equation */
14  {
15      int x,t;                        /* Counters in array for solution */
16      double u[TN+1][XN+1];
17
18      for (t=1;t<=TN;t++)      /* Boundary values:  u(t,0)=u(t,1)=0 */
19      u[t][0]=u[t][XN]=0;
20
21      for (x=0;x<=XN/2;x++)       /* Initial values */
22      u[0][x]=2*x/XN;
23      for (;x<=XN;x++)      /* Initial values */
24      u[0][x]=2-2*x/XN;
25
26      for (t=0;t<=TN-1;t++)      /* Explicit difference equation approximation */
27      for (x=1;x<=XN-1;x++)
```

```
28          u[t+1][x]=r*u[t][x-1]+(1-2*r)*u[t][x]+r*u[t][x+1];
29      }
```

The values of "*r*" (in line 4) and "*dx*" (in line 5) can both affect the space-quality trade-off. With a larger "*r*" or "*dx*", the algorithm generates an approximate solution of the heat equation with less accuracy but the solution use less memory space. With a smaller "*r*" or "*dx*", the effect is reversed.

3. COMPONENT PARAMETERIZATION PROCESS

First, consider how to manually customize a component to satisfy different QoS requirements. The designer first needs to go through the source code of the component and identify all of the statements that can impact the QoS trade-offs. Then the designer needs to modify these statements to adapt the component's QoS trade-offs. These modifications should not change the component's functional capabilities. The impacts of these modifications on the components' QoS trade-offs are measured and recorded. Finally, when the designer actually uses the components to develop a system, he can determine a specific modification to customize the components according to the recorded measurement data and the QoS requirements the component should satisfy. There are three major weaknesses inherent in this manual approach. First, it is time consuming to manually go through any reasonable size components, especially for designers who are not familiar with the components. Second, designers may miss statements that can impact QoS trade-offs. Third, designers need substantial training before they are experienced in identifying the numerous kinds of potential statements that can impact QoS trade-offs. These weaknesses motivate us to develop a semi-automated approach that can aid in the systematic identification and modification of statements to achieve QoS trade-offs.

When transforming the manual process to an automated approach, a rule base is a flexible and extensible way to implement the process. Since there may be many potential rules for configurable parameter identification, it is not easy to identify a complete set of rules a-priori. New rules can be introduced conveniently with a rule base. Furthermore, different rules may be applicable for different application domains. As a result, the users may have different QoS tradeoff goals and may wish to apply specific sets of rules for individualized identification processes. The use of the rule base can

provide flexibility in rule set selection, as well as help to organize and manage the rules.

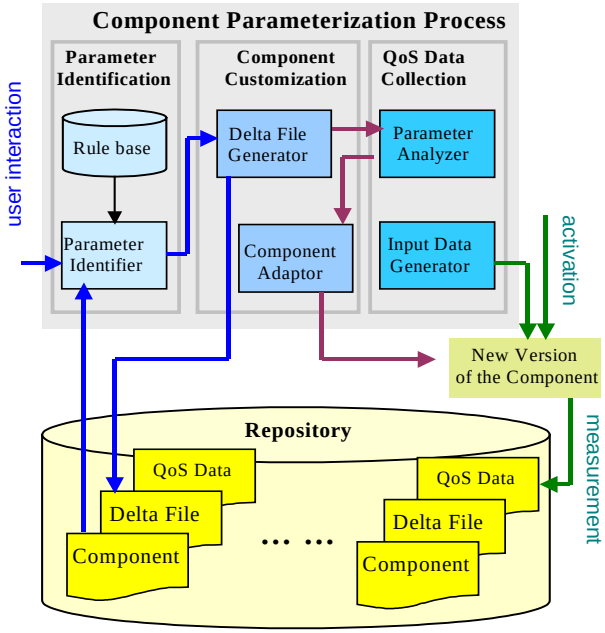

Figure 1. Overview of component parameterization technique.

The rule-based component parameterization process presented here is divided into three steps (refer to Figure 1). The first step is to identify configurable parameters in a non-QoS-R component that can be parameterized to achieve QoS tradeoffs. The second step is modifying the non-QoS-R component using the delta files according to each configurable parameter identified so that the component becomes a QoS-R component. These QoS-R components can also be reconfigured later to satisfy different QoS requirements of the system under development. Finally, QoS data of each QoS-R component under different configurations are collected and stored in the repository. These QoS data record the impact of configurable parameters to QoS properties and provide necessary information for component retrieval and composition analysis in future system development.

3.1. Configurable Parameter Identification

The first step of configurable parameter identification process is to analyze the source code of a component. The analysis is based on the grammar of the programming language. After the first step, the source code of the component is converted to a parse tree. This step is quite similar to what the front end of a traditional compiler does. It can be divided into three passes, which communicate with one another. The process model is shown in Figure 2. The first pass is the preprocessor which does macro substitutions, strip comments from the source code, etc. The preprocessor in the configurable

parameter identification process is slightly different from the preprocessor of a C compiler. The difference exists in the handling of the macro definition. Definitions are indicated by keyword "*define*" in C language. They consist of a name for the macro being defined and a body, forming its definition. If the body of the definition is a single number, then the preprocessor of configurable parameter identification treats the macro as a special case of constant variable declaration and does not do macro substitutions. This approach is necessary for identifying loop control variable in Newton's example. Using the preprocessor of the compiler, after preprocessing, the "*epsilon*" in Newton's example is substituted by the value "0.001". It is not known whether "0.001" is from macro definition or is only a constant in later phase. Hence, "*epsilon*" cannot be identified as a loop control variable. The second pass in the process consists of lexical analysis and syntactic analysis. It translates the C source code into an abstract syntactic tree (AST). The last pass extracts group of facts from AST based on fact analysis rules. These facts are then analyzed based on configurable parameter identification rules and the potential configurable parameters are identified.

The configurable parameter identification process is not finished after the third pass. The results from the third pass needs to be checked by software designers who are domain experts. Some identified configurable parameters which are not real configurable parameters in the specific component's context or not useful for the user are discarded.

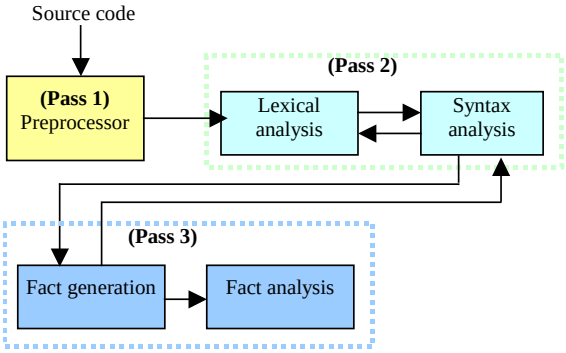

Figure 2. Process of configurable parameter identification.

3.2. Delta file

A QoS-R component, as defined in Section 3, is the combination of the original component and a delta file. The delta file stores a set of adaptation specifications. Each adaptation specification includes a configurable parameter's name, measured QoS data, and an edit script.

The edit script specifies how to parameterize a configurable parameter in the form of deltas, i.e., as differences between files. The edit script consists of edit commands which have the form "n1 n2 command text". Here, "n1" and "n2" represent lines in the file. Identical pairs, where n1=n2 are abbreviated as a single number n1. The "command" can be one of three command types: append, delete, and replace. The append command accepts zero or more lines of text and appends it after the addressed line. The delete command deletes addressed lines. The replace command deletes the addressed lines and accepts zero or more lines of text that replaces these lines. The "text" is zero or more lines of text used in the append and change commands.

The configurable parameters are classified into two categories according to the ways they are customized. The first category includes configurable parameters whose customizations only require change of their initial values. Such category includes the loop control variable and the array size parameter. The second category includes configurable parameters whose customizations require changes to some code segments, such as random number generator example given in Section 2. For the first category, the QoS data records the relationship between the initial value of the configurable parameter and the QoS attributes it impacts. This relationship is generally expressed by functions which are acquired at QoS data collection step. For example, the delta file of Newton's root computation program may have two functions. One function describes the relationship between running time of the component and the '*epsilon*' value. Another function describes the relationship between precision level of the root value and the '*epsilon*' value. For the second category, the QoS data records the QoS attribute values corresponding to all possible customizations on the component. These data are also acquired at QoS data collection step. The QoS data defined in a delta file is used at the time of component customization. The actual value of a configurable parameter or the customization scheme is determined after analyzing QoS data and QoS requirements. For example, given the expected running time and precision level of the Newton's root computation program, two '*epsilon*' values can be calculated from two functions defined in QoS data. Then the average of these two values is a good choice to customize the component.

The delta file can be generated semi-automatically from identified configurable parameters. Generating the delta file for a component which only has loop control variable configurable parameter

is straightforward. The delta file generator can simply use the line information stored together with the variable in the symbol table to write delta file. However, how to automatically generate delta file for a component which has configurable parameters like random number generator example is a more challenging task. It requires that the delta file generator can analyze the program structure and logic of the component and modify its structure and logic without changing the functional capability of the component. Future research is going to address this problem.

3.3. QoS Data Collection

The impact of the configurable parameters to QoS attributes should be measured and stored in the repository in order to allow the proper setting of the configurable parameters for a given execution environment and system requirement. The set of QoS attributes measured for each configurable parameter is application dependent. The units of these QoS attribute measurements also depend on the application. There are three kinds of QoS attribute measurements which are corresponding to three kinds of trade-offs: 1) Running time, measured in microseconds, milliseconds, CPU cycles, etc. 2) Memory usage, measured in bytes, kilobytes, etc. 3) Output quality, measured in accuracy or precision level, etc. For each component with configurable parameters, two elements need to be determined before QoS data collection activity starts. One element is the parameter space which defines the set of values which is reasonable for a configurable parameter. Another element is the input domain which defines the complete set of potential inputs of the component. Typically, it is not practical - or even possible - to test the component with all potential inputs. Hence, the idea of the equivalence class test is used here to solve the difficulty. The whole input domain is partitioned into a number of equivalence classes. A single input is selected from each equivalence class with assumption that all inputs in the same equivalence class generate roughly same relationship between configurable parameter and QoS attributes. The situation for parameter space is similar. The parameter space is partitioned into a number of equivalence classes for a given input. A single value is selected from each equivalence class and the resulted group of values is used as the test values for the input. The test results of the group of values are used to construct an approximate function which describes the impact of the configurable parameter to QoS attribute for the given input. Some QoS

data collection activities require the execution of component (e.g., running time measurement). However, most components cannot execute standalone. Therefore, a simple driver needs to be constructed manually to trigger the component. The driver is also stored in repository with the component.

4. RULE-BASED CONFIGURABLE PARAMETER IDENTIFICATION PROCESS

Configurable parameter identification is an important and challenging part of the parameterization process. The rule-base plays an important role during the identification process. Two sets of rules are defined to assist the identification process, one set for fact analysis and one set for configurable parameter identification. The fact analysis rules are used during analysis of the source code of a component. These rules extract syntactic and semantic facts of grammar construct from the program. After the entire program is scanned and associated facts are extracted, then the configurable parameter identification rules are used to evaluate every variable and constant identifier declared in the program to check if a variable or constant is a specific kind of configurable parameter. The configurable parameter identification rules are stored in the rule base to facilitate future updates and extensions. The rule base provides a flexible and extensible approach to update and manage the rules. Decoupling the fact analysis rules and configurable parameter identification rules also allows the configurable parameter identification rules to be independent of the programming languages in use, thus, can be easily adapted to support multiple languages. The fact analysis rules do not need to be stored in the rule base since they rarely need to be updated after they are defined, unless the target programming language gets updated. These rules are embedded into the fact generation module in the tool support.

4.1. Fact Analysis Rules

The basic facts in the program can be classified into identifiers, statements, and expressions. Identifiers are the basic data objects of a program. Some of them may be configurable parameters. For example, a static or global array can affect memory requirement. The space allocated to a static or global array is not released until the execution of the program is finished. If the static or global array is replaced with a local array declared in a function, then the memory space is reclaimed after the

execution of the function is finished. However, since a local array cannot save the array's values across function's execution, the array's values are reconstructed every time at the beginning of a function's execution. This increases the program's execution time. An array is defined by identifier in a programming language. An identifier can also affect the execution time of a program. For example, in most cases, how many times the loop body of a loop statement is executed is controlled by the loop guard condition (break, goto statement, etc. can affect it). A loop's guard condition is generally a comparison expression. The variables and/or constants appear in the comparison expression decides the comparison result and hence decides the execution of loop body. A fact is used to express that an identifier is of a certain type. More specifically, ***constant*(*d*)** and ***variable*(*d*)** indicate that the identifier ***d*** is a constant or variable. ***array*(*d, e*)** indicates that identifier ***d*** is an array of size ***e***. Note here ***e*** may be a number or expression. ***static*(*d*)** indicates that the storage type of identifier ***d*** is static. ***global*(*d*)** indicates that the identifier ***d*** is declared outside of all functions. A detailed list of facts for identifiers can be found in [23].

A statement is the basic execution unit of a program. A variety of different kinds of statements need to be identified in a program. For example, a variable's value is generally changed in assignment statement. Hence, assignment statements need to be identified in a program in order to know where a variable's value is changed. Another example is a loop statement. Loop statements dominate the execution time of a program. They are potential statements that can affect the program's execution time versus quality or space and also need to be identified. A fact is used to express that a statement is of a certain type. More specifically, ***assignment-statement*(*s, lhs, rhs*)** indicates that statement ***s*** is an assignment statement where ***lhs*** and ***rhs*** are the left hand side expression and the right hand side expression of ***s*** respectively. A ***loop-statement* (*s, c, b*)** indicates that statement ***s*** is a loop statement (i.e., ***s*** is one of for, while, do while statement) where expression ***c*** is the loop condition of ***s*** and compound statement ***b*** is the loop body of ***s***.

An expression consists of identifiers and operators and is the constituent element of the statement. During configurable parameter identification, it is necessary to know where an expression appears and what type of an expression it is. For example, a variable's value can also be changed in unary increment or unary decrement expression (i.e., an expression such as "*i*++") in addition to assignment

statement. Here the unary operator "++" and "– –" can appear before or behind the variable. Hence, the information about which variable's value is changed can be got from the unary increment or unary decrement expression. Another example is comparison expression. How many times the loop body of a loop statement is executed is controlled by the loop guard condition. This loop guard condition is generally a comparison expression. A fact is used to express that an expression is of a certain type. More specifically, ***expression(e)*** indicates that ***e*** is a general expression. ***unary-inc-expression*(*e*)** and ***unary-dec-expression*(*e*)** indicate that the expression ***e*** is a unary increment or unary decrement expression. Note the case which unary operator appears before or after the variable is not distinguished. A ***comparison-expression*(*e, lhs, rhs*)** indicates that the expression ***e*** is a comparison expression (i.e., ***e*** is one of less than, greater than, less than or equal, greater than or equal, equal, not equal expression). Here ***lhs*** and ***rhs*** are the left hand side expression and the right hand side expression of ***e*** respectively.

A fact is also defined to express the relationship between identifiers, expressions and statements. We use ***inside*(*d, e*)** to express that grammar construct ***d*** appears in grammar construct ***e***. Note here ***d*** may be a constant, variable or statement, and ***e*** may be an expression, statement or a compound statement.

It is unnecessary here to show the complete facts defined in the technique. A detailed list of facts can be found in [23]. The defined facts cover all the terminal and non-terminal symbols in the grammar definition of ANSI C. Currently not all of these facts are used in configurable parameter identification rules. However, these facts may be used in the future when the set of configurable parameter identification rules are extended.

4.2. Configurable Parameter Identification Rules

A set of configurable parameter identification rules is defined to identify different kinds of configurable parameters in a program. The configurable parameter identification rules use the basic facts extracted from the program by fact analysis rules as premises to generate conclusions. These conclusions express if a constant or variable in the program is a configurable parameter.

Here three examples of configurable parameters are presented: loop control variable, static-or-

global array and array size parameter. Loop control variables generally impact the program's time-quality trade-off. This conclusion comes from the observation of numerical analysis examples studied so far. A larger loop control variable can improve the result's quality at the cost of increasing program execution time. A smaller loop control variable reverses the effect. The rule for identifying a loop control variable is as follows: for each loop statement, if the guard condition of loop statement is a comparison expression with a constant value on left hand side of inequality and some variables on right hand side of inequality (or vice versa) and if in body of loop there exists the same variables on left hand side of assignment statement, then the constant value on one side of inequality is a loop control variable. This can be defined as a predicate ***loop-control-variable(d)***:

(∀d ∃d1 ∃s ∃c ∃b ∃lhs1 ∃rhs1 ∃s1 ∃lhs2 ∃rhs2)
(**loop-control-variable(d)** ⇒
constant(d) ∧
variable(d1) ∧
loop-statement(s, c, b) ∧
comparison-expression(c, lhs1, rhs1) ∧
(inside(d, lhs1) ∧
inside(d1, rhs1) ∨
inside(d1, lhs1) ∧
inside(d, rhs1)) ∧
assignment-statement(s1, lhs2, rhs2) ∧
inside(s1, b) ∧
inside(d1, lhs2))

A rule is also defined to identify static or global array which may impact time-space trade-off. The rule for identifying a static or global array is as follows: if an identifier ***d*** is an array of size ***n*** which is larger than ***k*** and the storage type of the array is static or the array is declared outside of all functions and if there exists the same array ***d*** on both sides of an assignment statement, then the identifier ***d*** is a static or global array. Note here ***k*** is a threshold to control how large of an array's size can have significant impact on the program's memory requirement property. The rule can be defined as a predicate ***static-global-array(d)***:

(∀d ∃n) (**static-global-array(d)** ⇒
variable(d)∧
array(d, n)∧
(n>k)∧
(static(d)∨
global(d))∧
assignment-statement(s, lhs, rhs)∧
inside(d, lhs)∧

inside(d, rhs))

A rule is also defined to identify array size parameter which may impact Space-quality trade-off. The rule for identifying an array size parameter is as follows: if an identifier is an array and the size of the array is an expression which includes a constant identifier, then the constant identifier is array size parameter. The rule can be defined as a predicate ***array-size-parameter*(*d*)**:

(∀d ∃d1 ∃e) (**array-size-parameter(d)** ⇒
constant(d)∧
variable(d1)∧
array(d1,e)∧
expression(e)∧
inside(d, e))

4.3. Example

Newton's algorithm is used to illustrate how the fact analysis rules and configurable parameter identification rules are used. The fact analysis rules are applied first during the top down scan of the program (refer to Figure 3). The grammar construct facts extracted are summarized as follows.

- The fact that identifier ***epsilon*** is a constant is obtained from line 2 "*#define epsilon* 0.001". This is expressed by ***constant*(*epsilon*)**.
- The fact that identifier ***xnew*** is a variable is obtained from line 6 "*float xold, xnew;*". This is expressed by ***variable*(*xnew*)**.
- The fact ***assignment-statement*(*s*1, *l*2, *r*2)** is obtained from line 16, "*xnew = xold–*(*f*(*xold*))/(*df*(*xold*));" which is a assignment statement. Here ***s*1** refers to the statement. ***l*2** is the left hand side expression "*xnew*" of ***s*1** and ***r*2** is the right hand side expression "*xold–*(*f*(*xold*))/(*df*(*xold*))".
- The fact ***loop-statement*(*s, e, b*)** is derived from the code "*do* {…} *while* (*fabs*(*xold–xnew*)>*epsilon*)" between line 12 and line 18, which as a whole is a do-while statement. Here ***s*** refers to this do-while statement, ***e*** is the loop guard condition "*fabs*(*xold–xnew*)>*epsilon*" and ***b*** corresponds to the compound statement between keywords "*do*" and "*while*".
- The fact ***comparison-expression*(*e, l*1, *r*1)** is derived from line 18, the loop guard condition "*fabs*(*xold–xnew*)>*epsilon*". Here ***e*** refers to the condition. ***l*1** refers to the left hand side expression "*fabs*(*xold–xnew*)" of ***e*** and ***r*1** corresponds to the right hand side expression "*epsilon*" of ***e***.

Additionally, some relationship facts are also extracted and shown as follows.

- The assignment statement ***s*1** is a statement within compound statement ***b***, hence the fact ***inside*(*s*1, b)**.
- The identifier ***xnew*** appears in the left hand side expression ***l*2** of the assignment statement ***s*1**. The fact ***inside*(*xnew*, *l*2)** is obtained from it.
- The identifier ***epsilon*** appears in the right hand side expression ***r*1** of the loop guard condition ***e***. The identifier ***xnew*** appears in the left hand side expression ***l*1** of ***e***. Thus, the facts ***inside*(*epsilon*, *r*1)** and ***inside*(*xnew*, *l*1)** are obtained.

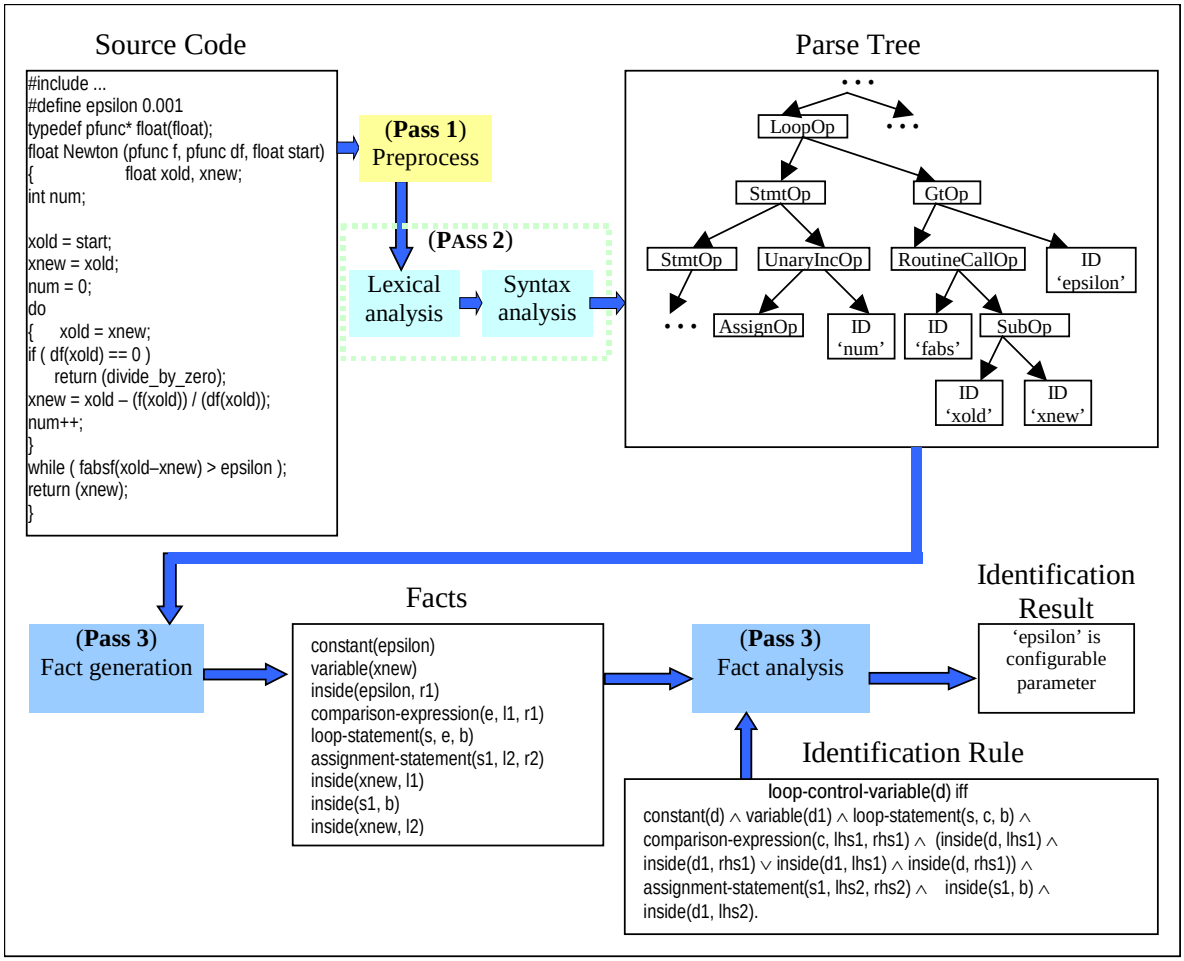

Figure 3. Facts from the Newton's example.

Note that facts described above do not represent the complete set of facts that can be obtained from Newton's example. For example, the fact ***variable*(*num*)** from line 7 "*int num*;" can also be obtained. However, the above facts are enough to derive a conclusion from the configurable parameter identification rules in this example.

When the fact extraction finishes, the configurable parameter identification rules are used to

analyze extracted facts and generate conclusions. The first rule used is the **loop-control-variable** rule. When the rule is applied to on the identifier ***epsilon***, the rule finds that all the premises for the rule are true and it gives the conclusion that ***epsilon*** is a loop control variable. The second rule applied is the ***static-global-array*** rule. However, since no array variables exist in the example, the evaluation fails on all identifiers.

5. TOOL SUPPORT

A complete tool set to cover the whole process of component parameterization is being developed. The tool set consists of three modules which correspond to three step approach of component parameterization technique respectively. The parameter identifier is the major tool for the parameter identification process. The component customization module consists of two tools, the delta file generator and the component adaptor. Since the delta file generator needs to access the parse tree and symbol table, it is integrated into the parameter identifier. The configurable parameters selected by the parameter identifier are passed to the delta file generator and the delta file generator then generate the specific delta file format; the delta file is stored in the repository where it is associated with its original component. The function of component adaptor is to generate a new version of the component based on a specific set of values for the configurable parameters in a component. The set of values can be generated by an analyzer or provided by the user.

When a component is used to compose a system, the settings of the configurable parameters are going to depend on the execution environment (static as well as dynamic), the QoS requirements of the application system, and the parameter settings of other components. The Composition Analyzer in ORES is used to perform the analysis and determine the parameter settings. Tools supporting test and QoS Data collection for each component are highly desired to facilitate the composition analysis. The tools in the QoS Data Collection module include the parameter analyzer and input data generator. Component specification and the QoS analysis objectives should be provided as inputs to the QoS data collection process. Each set of parameter settings generated by the parameter analyzer is sent to the component adaptor to generate a new version of the component. The new version is then invoked using the input data generated by input data generator and its QoS data are collected. The collected

QoS data are also stored in the repository for future use.

Currently, the parameter identifier which supports the partial automation of configurable parameter identification is being developed. The tool consists of three parts, the GUI, engine, and fact analyzer. The GUI interacts with the user directly, accepts the user's input and displays the output. The engine parses the source code of a component and extracts facts from its grammar constructs. The fact analyzer analyzes these facts and identifies potential configurable parameters of the component. The system design in UML is shown in Figure 4.

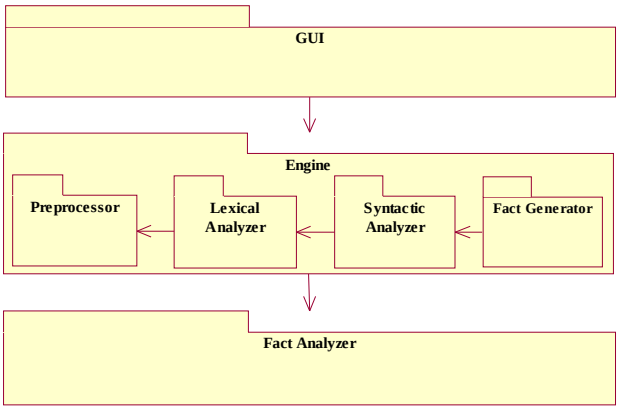

Figure 4. System design of the parameter identifier.

The tool is programmed using a combination of Java, C and Prolog. The GUI is implemented in Java. The Engine is implemented in C. The Fact Analyzer is implemented in Prolog. Furthermore, Lex and Yacc are also used to develop the engine. This design decision comes from two considerations: (1) GUI development. Using Java to develop GUI is a convenient, fast approach; (2) The portability. The tool set needs to be ported easily to other platforms such as Windows and Macintosh. Java is more portable in comparison to other languages. Furthermore, the C code and Prolog code are also convenient to port when they only use ANSI features.

The main functions of the Parameter Identifier are similar to a front end of a conventional compiler: first performs preprocess on the component, then performs lexical and syntactic analysis and builds the parse tree for the component. These functions are partly implemented by using Lex, Yacc and standard C grammar.

5.1. Preprocessor, Lexical Analyzer and Syntactic Analyzer

The preprocessor uses a modified preprocessor for the C compiler. The modification exists in macro definition. For macro definitions which have the form of '#define NAME number" where NAME is the macro name and number is a constant value, the preprocessor of parameter identifier do not process them and leave them to lexical analyzer and syntactic analyzer. The lexical analyzer is generated from Lex. The syntactic analyzer is generated from Yacc.

5.2. Fact Generator

The result from syntactic analyzer is an abstract syntax tree (AST). A binary tree data structure is used to express the AST. Each node in the AST corresponds to a grammar construct (e.g., identifier, expression, and statements) in the program and is assigned a unique number as reference. The facts are then generated from these nodes by recursively traversing the AST. The facts are expressed as terms of the Prolog language. For example, the term ***loopOp*(*x*1, *x*2, *x*3)** expresses the fact that node ***x*1** is a node corresponds to a loop statement and node ***x*2**, ***x*3** are node ***x*1**'s left child node and right child node respectively. The left child node of the loop statement node corresponds to the loop statement's loop guard condition in AST. The right child node corresponds to the loop body. Note ***x*1**, ***x*2**, and ***x*3** are the unique numbers assigned to each of these three nodes (this convention is used in other terms). The term ***loopOp*(*x*1, *x*2, *x*3)** corresponds to ***loop-statement*(*s*, *c*, *b*)** defined in fact analysis rules. The term ***idnode*(*x*1, *id*, *type*)** is used to express that node ***x*1** is a node denotes an identifier where ***id*** is the string literal of the identifier and ***type*** is either "const" or "var". The term ***idnode*(*x*1, *id*, *type*)** corresponds to ***constant*(*d*)** or ***variable*(*d*)** defined in fact analysis rules depends on ***type***. A node corresponds to comparison expression is expressed by one of several terms depending on the type of comparison operator. Such terms include ***gtOp*(*x*1, *x*2, *x*3)**, ***ltOp*(*x*1, *x*2, *x*3)**, ***geOp*(*x*1, *x*2, *x*3)**, and ***leOp*(*x*1, *x*2, *x*3)**. Note here node ***x*1** is the node corresponds to a comparison expression and ***x*2**, ***x*3** are node ***x*1**'s left child node and right child node. The left child node of the comparison expression node corresponds to the left hand side expression of the comparison expression in AST. The right child node corresponds to the right hand side expression of the comparison expression. These four terms together correspond to ***comparison-statement*(*e*, *lhs*, *rhs*)** defined in fact analysis

rules. A node corresponds to assignment statement is expressed by term ***assignOp*(*x*1, *x*2, *x*3)**. Here node ***x*1** is the node corresponds to an assignment statement and ***x*2**, ***x*3** are node ***x*1**'s left child node and right child node respectively. The left child node of the assignment statement node corresponds to the left hand side expression of the assignment operator in AST. The right child node corresponds to the right hand side expression. The term ***assignOp*(*x*1, *x*2, *x*3)** corresponds to ***assignment-statement(s, lhs, rhs)*** defined in fact analysis rules. Moreover, a term ***childNode*(*x*1, *x*2)** is defined to express the parent-child relationship between nodes in AST. This term means node ***x*1** is either left child node or right child node of node ***x*2**. The term ***childNode*(*x*1, *x*2)** together with the predicate ***ancestor*(*x*1, *x*2)** defined in fact analyzer corresponds to ***inside*(*d, e*)** defined in fact analysis rules.

Note here only part of the terms defined in the tool is shown. The complete set of terms covers the entire set of facts defined in fact analysis rules. However, show the complete set of terms occupies much space and gives no more help to understand the rule-base. Hence, this paper only uses the subset to give the reader a sense of the terms and to illustrate how fact analyzer is implemented.

5.3. Fact Analyzer

The configurable parameter identification rules are implemented as Prolog predicates. These predicates are based on the terms defined in fact generator. For example, the identification rule corresponding to predicate ***loop-control-variable*** is defined as the following Prolog predicate:

```
loop_control_variable(ConstId):-
        loopOp(_, TestNode, LoopBodyNode),
        idnode(ConstCode1, ConstId, const),
        idnode(VarCode1, VarId, var),
        (gtOp(Node, LeftNode, RightNode);
        ltOp(Node, LeftNode, RightNode);
        geOp(Node, LeftNode, RightNode);
        leOp(Node, LeftNode, RightNode)),
        ancestor(Node, TestNode),
        (ancestor(VarCode1, LeftNode),
        ancestor(ConstCode1, LeftNode));
        (ancestor(VarCode1, RightNode),
        ancestor(ConstCode1, RightNode)),
        assignOp(AssignNode, Left, _),
        idnode(VarCode2, VarId, var),
        ancestor(VarCode2, Left),
        ancestor(AssignNode, LoopBodyNode).
```

Here ***ancestor*** is an ancillary predicate to express the recursive inside rule.

ancestor(X, Y):- *inside*(*X, Y*)**.**
ancestor(X, Y):- *inside*(*Z, Y*), *ancestor*(*X, Z*).

The identification rules in Prolog predicates are stored in a separate file and implemented as an independent Prolog program. This prolog program constitutes the fact analyzer as well as the rule base. The addition and update of the rule base is accomplished by simply adding new predicates or modify existing predicates in this file. The modification of this file does not affect other modules (i.e., GUI, engine) in the parameter identifier. This provides great flexibility and extensibility to the tool. During the execution of the parameter identifier, this prolog program is loaded dynamically and analyzes the extracted facts from AST. The analysis results are returned to the GUI part.

6. APPLICATION OF THE COMPONENT PARAMETERIZATION TECHNIQUE

An example scenario using the parameter identifier tool is described here. During the execution, the user first needs to configure the path of the executable program and then opens the C source file of a component. Before executing the request to identify configurable parameters, the user can configure which configurable parameter identification rules they want to use. This significantly reduces the execution time in comparison to selecting all the rules to use. If the user wants to handle time, space and quality trade-offs at the same time, then he/she can choose to use all of the rules. At last, the user executes “identifying configurable parameters” menu item and gets the identification results. The identification result is shown in the output window. Furthermore, the line number where the configurable parameter declared is also shown to enable user locate the configurable parameter quickly. For each identified configurable parameter, the tool also provides a recommendation for the user on how to utilize the identification result to customize the component.

Three examples given in Section 3 are used here to validate the component parameterization technique and the tool support (currently the parameter identifier). The component parameterization result on Newton’s algorithm is shown in Figure 5. The parameter identifier identifies loop control variable “***epsilon***” in Newton’s algorithm and highlights the corresponding line in the code window. Moreover, the parameter identifier provides a suggestion on how the configurable parameter affects the time-quality trade-off.

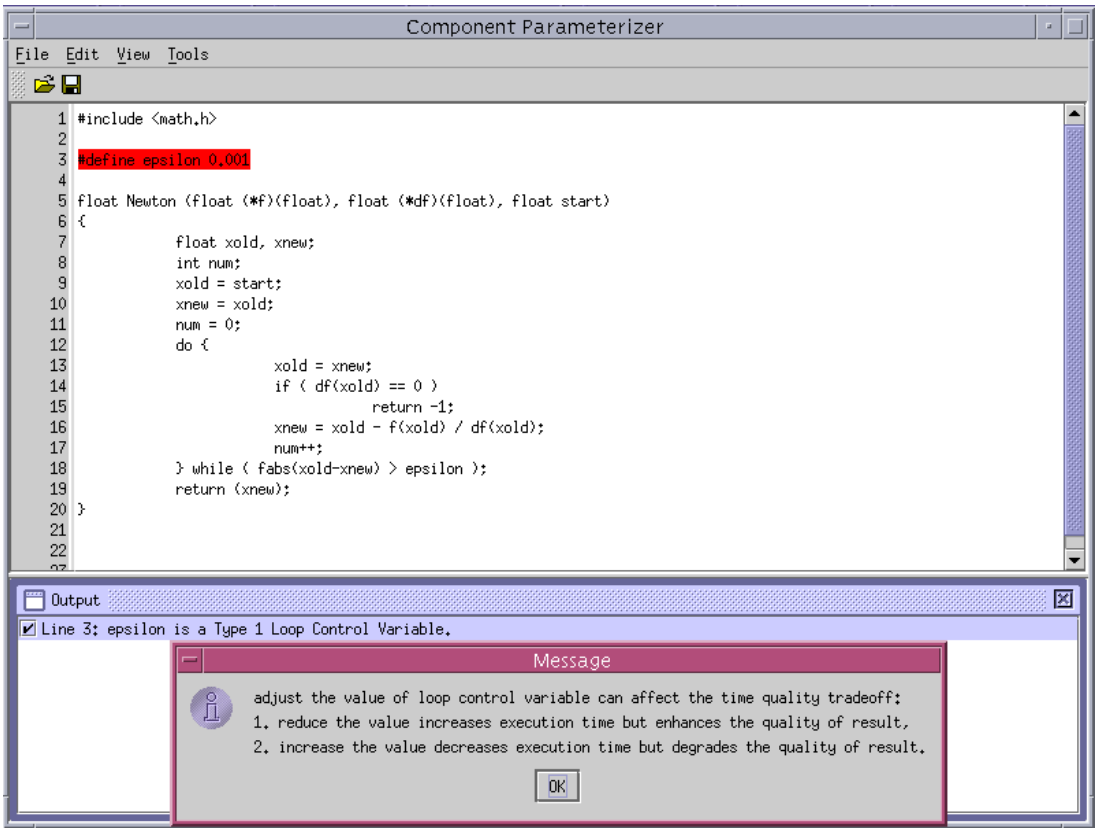

Figure 5. Parameter identification result of Newton's example.

A manual customization result of the Newton's algorithm based on this suggestion is shown in TABLE IV. The modification is on line 2 where the initial value of "epsilon" is changed to 0.000001 in order to favor precision over running time. Note here the initial value is chosen arbitrarily to show a possible customization. The chosen initial value is decided by the QoS requirement of target system and QoS data collection result on this component during actual system development. The other two examples also use the same way to demonstrate the customization.

TABLE IV. NEWTON'S ALGORITHM AFTER CUSTOMIZATION

```
1     #include ...
2     #define epsilon 0.000001
...   ... /* same as original code */
```

The component parameterization result on random number generator is shown in Figure 6. The parameter identifier determines that the array variable "***mt***" is a configurable parameter of type static-global-array. Moreover, the parameter identifier shows a suggestion on how the configurable parameter affects the time-space trade-off.

A customization result based on this suggestion is shown in TABLE V. Current seed values are stored in a local array *mt*[*N*] declared in function *genrand*() at line 31. The initial seed is store in a

static variable *mt_seed* in function *sgenrand*() at line 25. The modified *genrand*() computes the values of the array *mt*[*N*] first (line 39-41) before computing the random number from *mt*[*N*]. For every *N* calls, the initial seed stored in static variable *mt_seed* is updated (line 42-55).

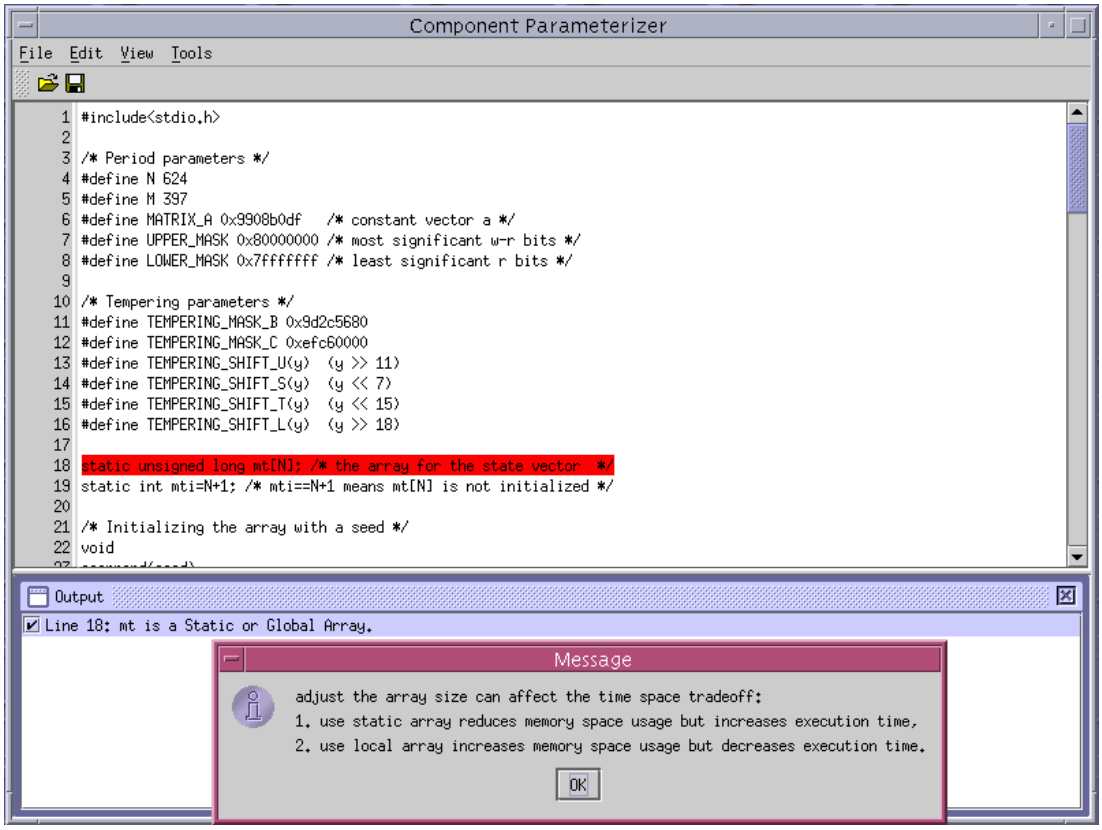

Figure 6. Parameter identification result of random number generator example.

TABLE V. RANDOM NUMBER GENERATOR AFTER CUSTOMIZATION

```
1     #include <stdio.h>
…     ….
18    static unsigned long mt_seed; /* the initial seed  */
19    static int mti=N+1; /* mti==N+1 means mt[N] is not initialized */
20
21    /* initializing the array with a NONZERO seed */
22    void sgenrand(unsigned long seed)
23    {
24       /* setting initial seeds to mt[N] */
25        mt_seed = seed & 0xffffffff;
26        mti = N;
27    }
28
29    unsigned long genrand()
30    {
31       unsigned long mt[N];
32       unsigned long y;
33       static unsigned long mag01[2]={0x0, MATRIX_A};
34       /* mag01[x] = x * MATRIX_A  for x=0,1 */
35
36       if (mti == N+1)   /* if sgenrand() has not been called, */
37          sgenrand(4357); /* a default initial seed is used   */
38       int kk;
39       mt[0]= mt_seed;
40       for (kk=1; kk<N; kk++)
41          mt[kk] = (69069 * mt[kk–1]) & 0xffffffff;
42       if (mti == N) { /* generate N words at one time */
…          generate new values of mt[0..N–1]
54          mt_seed = mt[0];
```

```
55      }
56      ...
57  }
```

The parameterization result on solving heat equation example is shown in Figure 7. The parameter identifier identifies "*r*" and "*dx*" as configurable parameters. Moreover, the parameter identifier shows a suggestion on how the configurable parameter affects the Space-quality trade-off.

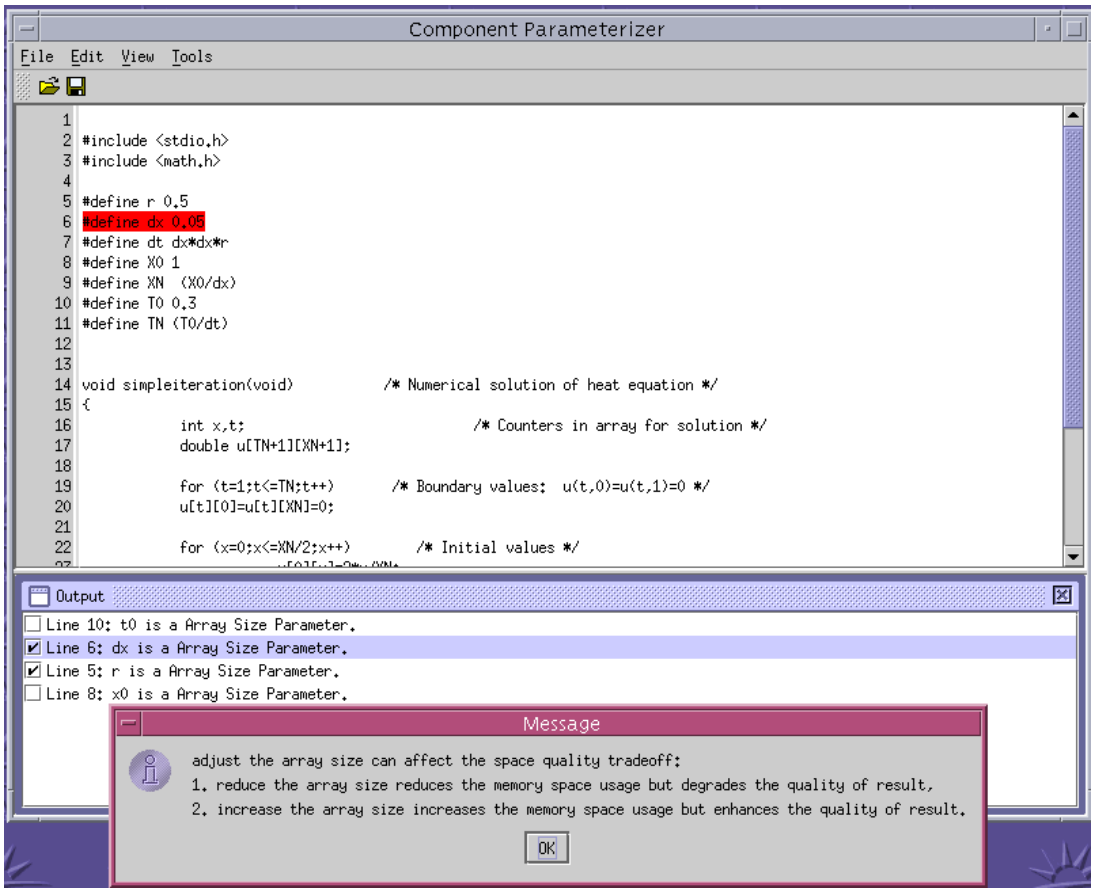

Figure 7. Parameter identification result of solving heat equation example.

A customization result based on this suggestion is shown in TABLE VI. The modification is on line 4 where the initial value of "***r***" is changed to 0.2 in order to favor precision over memory.

TABLE VI. SIMPLE ITERATION SOLUTION OF HEAT EQUATION AFTER CUSTOMIZATION

```
1   #include <stdio.h>
2   #include <math.h>
3
4   #define r    0.2
5   #define dx   0.05
6   … /* same as original code */
```

The above customization solutions show only one of the possible solutions to accomplish their trade-offs. There may exist other customization solutions for these examples.

7. CONCLUSION

This paper presents a component parameterization technique which aims at customizing non-QoS-R components into QoS-R components. These QoS-R components can be used to build embedded systems with different QoS requirements. They can also be used to develop adaptive embedded systems which are working in dynamically changing environment.

The tool support developed for the component parameterization technique enables software engineers to carry out the component parameterization process semi-automatically. The rule based approach introduced in configurable parameter identification provides flexibility and extensibility. Software engineers can conveniently add, delete and modify rules to achieve the desired identification goals. Moreover, the rule based approach supports module separation and improves the maintainability of the Parameter Identifier tool support.

Automated configurable parameter identification may have some limitations. Generally, it is difficult for the configurable parameter identification tool to identify all the desirable configurable parameters of a component. For example, the tool may identify configurable parameters that are not useful or not desired but miss some important configurable parameters. The tool may also identify some configurable parameters that are desired in term of the trade-offs but not useful in the current application. Such problems can be partly improved by introducing more rules and adding more specific conditions in each rule. Furthermore, the component parameterization technique requires that the source code of a component is available. For a component in binary code (e.g., a dynamically-linked shared library), the technique cannot be applied.

There are several research directions following this work. First, a large number of embedded software source code examples are going to be collected. The examples are going to be investigated and analyzed to identify additional configurable parameter identification rules and construct a comprehensive rule base. Furthermore, these examples are going to be used as test cases for validation of the rule base and tool support to validate the effectiveness of the approach. Second, the problem of collecting and effectively using QoS data to improve the component parameterization process needs to be investigated. The research problem here involves defining what QoS data is collected, how to

store the QoS data together with associated component, how to make use of these QoS data during composition analysis, and how to match the composition analysis result with each configuration of the QoS-R component. As this work progresses, the tool support is also going to be extended.

REFERENCES

[1] Brown, A. and Barn, B. (1999) Enterprise-scale CBD: building complex computer systems from components. *Proceedings of Software Technology and Engineering Practice* (STEP'99), Pittsburgh, PA, August, pp. 82–93. IEEE Computer Society Press, Washington, DC.

[2] Crnkovic, I. and Larsson, M. (2000) A case study: demands on component-based development. *Proceedings of the 22nd International Conference on Software Engineering* (ICSE'00), Limerick, Ireland, 04–11 June, pp. 23–31. ACM Press, New York, NY.

[3] Steward, D.B. and Khosla, P.K. (1991) Real-Time scheduling of dynamically Reconfigurable Systems. *Proceedings of the IEEE International Conference on Systems Engineering*, Dayton, OH, 1–3 August, pp. 139–142.

[4] Moser, M. (1996) Declarative scheduling for optimally graceful QoS degradation. *Proceedings of the IEEE International Conference on Multimedia Computing and Systems* (ICMCS'96), Hiroshima, Japan, 17–23 June, pp. 86–94.

[5] Baochun Li and Nahrstedt, K. (1999) Dynamic Reconfiguration for Complex Multimedia Applications. *Proceedings of the IEEE International Conference on Multimedia Computing and Systems* (ICMCS'99), Florence, Italy, 7–11 June, pp. 165–170.

[6] Schmidt, D.C. (1999) Middleware techniques and optimizations for real-time, embedded systems. *Proceedings of the 12th International Symposium on System Synthesis*, San Jose, CA, 10–12 November, pp. 12–16.

[7] Pyarali, I. Schmidt, D.C. and Cytron, R.K. (2003) Techniques for enhancing real-time CORBA quality of service. *Proceedings of the IEEE*, July, pp. 1070–1085.

[8] Pyarali, I.; Schmidt, D.C. and Cytron, R.K. (2002) Achieving End-to-end Predictability in the TAO Real-time CORBA ORB. *Proceedings of the 8th IEEE Real-Time and Embedded Technology and Applications Symposium*, San Jose, CA, 24-27 September, pp. 13-22. IEEE Computer Society Press, Washington, DC.

[9] Yau, S.S.; Yu Wang and Dazhi Huang (2003) Middleware Support for Embedded Software with Multiple QoS Properties for Ubiquitous Computing Environments. *Proceedings of the 8th International Workshop*

on Object-Oriented Real-Time Dependable Systems (WORDS'03), Guadalajara, Mexico, 15-17 January, pp. 250–256.

[10] Panahi, M.; Harmon, T. and Klefstad, R. (2003) Adaptive techniques for minimizing middleware memory footprint for distributed, real-time, embedded systems. *Proceedings of the 18th IEEE Annual Workshop on Computer Communications* (CCW'03), Dana Point, CA, 20-21 October, pp. 54–58.

[11] Geihs, K. and Becker, C. (2001) A framework for re-use and maintenance of Quality of Service mechanisms in distributed object systems. *Proceedings of the IEEE International Conference on Software Maintenance*, Florence, Italy, 7–9 November, pp. 470– 478.

[12] Becker, C. and Geihs, K. (2001) Quality of service and object-oriented middleware-multiple concerns and their separation. *Proceedings of the International Conference on Distributed Computing Systems Workshop*, Mesa, AZ, 16–19 April, pp. 117–122.

[13] Dongfeng Wang; Hui Ma; Bastani, F. and Yen, I.-L. (2004) Decomposition of fairness and performance aspects for high-assurance continuous process-control systems. *Proceedings of the 8th IEEE International Symposium on High Assurance Systems Engineering* (HASE'04), Tampa, FL, 25–26 March, pp. 3–11.

[14] Heineman, G.T. (1998) A Model for Designing Adaptable Software Components. *Proceedings of the 22nd Annual International Computer Science and Application Conference* (COMPSAC'98), Vienna, Austria, 19–21August, pp. 121–127.

[15] TRCS97-15 (1997) *Supporting the Integration and Evolution of Components Through Binary Component Adaptation*. Department of Computer Science, University of California, Santa Barbara, CA.

[16] Keller, K. and Hoelzle, U. (1998) Binary Component Adaptation. *Proceedings of the 12th European Conference on Object-Oriented Programming* (ECOOP'98), Brussels, Belgium, July, pp. 307–329. Springer-Verlag, London.

[17] Bosch, J. (1999) Superimposition: A component adaptation technique. *Information and Software Technology*, 41, 257–273.

[18] Brant, J.; Foote, B.; Johnson, R.E. and Roberts, D. (1998) Wrappers to the Rescue. *Proceedings of the 12th European Conference Object-Oriented Programming* (ECOOP'98), Brussels, Belgium, July, pp. 396–418. Springer-Verlag, London.

[19] Samentinger, J. (1997) *Software Engineering with Reusable Components*. Springer-Verlag, New York, NY.

[20] Cooper, K.; Jia Zhou; Hui Ma; Yen I.-L. and Bastani, F. (2003) Component parameterization for Satisfaction of QoS Requirements in Embedded Software. *Proceedings of ERSA'03*, Las Vegas, NV, 23–26 June, pp. 58–64.

[21] Jia Zhou; Cooper, K. and Yen I.-L. (2004) A Rule-Based Component Customization Technique for QoS Properties. *Proceedings of the 8th IEEE International Symposium on High Assurance Systems Engineering* (HASE'04), Tampa, FL, 25–26 March, pp. 302-303.

[22] Yen, I.-L.; Goluguri, J.; Bastani, F.; Khan, L. and Linn, J. (2002) A component-based approach for embedded software development. *Proceedings of the 5th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing* (ISORC'02), Washington, DC, April, pp. 402–412.

[23] UTDCS-50-03 (2003) *A Component Based Customization Technique for QoS Properties*. Department of Computer Science, University of Texas at Dallas, Richardson, TX.