A Generative Programming Framework for Adaptive Middleware*

Venkita Subramonian and Christopher Gill
Department of Computer Science and Engineering
Washington University, St.Louis,MO
{*venkita,cdgill*}*@cse.wustl.edu*

Abstract

Component middleware technologies such as the CORBA Component Model (CCM) [1], J2EE [2], and .NET [3], were developed to address many limitations like interdependencies between services and object interfaces, limited re-use, of first-generation middleware technologies such as CORBA 2.x [4], XML [5], and SOAP [6]. These component technologies have addressed a wide range of application domains, but unfortunately for distributed real-time and embedded (DRE) systems, the focus of these technologies has been primarily on functional and not quality of service (QoS) properties. Research on QoS-aware component models such as the CIAO project [7, 8] shows that there is a fundamental difference between configuration of functional and QoS properties even within such a unified component model: the dominant decomposition of functional properties is essentially object-oriented, while the dominant decomposition of QoS properties is essentially aspect-oriented. In this paper, we describe how a focus on aspect frameworks for configuring QoS properties both complements and extends QoS-aware component models.

This paper makes three main contributions to the state of the art in DRE systems middleware. First, it describes a simple but representative problem for configuring QoS aspects that cut across architectural layers, system and distribution boundaries, which motivates our focus on aspect frameworks. Second, it provides a formalization of that problem using first order logic - Infrastructure Configuration Logic - which both guides the design of aspect configuration infrastructure, and offers a way to connect these techniques with model-integrated computing [9] approaches to further reduce the programming burden on DRE system developers. Third, it describes alternative mechanisms to ensure correct configuration of the aspects involved, and notes the phases of the DRE system lifecycle at which each such configuration mechanism is most appropriate.

Keywords: adaptive and reflective middleware, system aspects, generative programming, first order logic.

1 Introduction

Constructing systems that are easily modified and extended is itself a challenging problem. A fundamental question is how to balance the rigor with which the correctness of any one configuration of the system can be assured, against the flexibility to evolve the system to other configurations that are also correct. Specifically, constraints applied during system development to ensure correctness often limit the range of adaptation that can be achieved at run-time.

We argue that for distributed real-time and embedded (DRE) systems the challenge of constructing correct yet evolvable systems is exacerbated, due to the additional challenges we outline in Section 1.1. While some of these challenges are readily addressed by specific modern development approaches such as the generative programming techniques described in Section 1.2, a complete framework for applying those techniques to component-based model-integrated development of DRE systems is needed.

1.1 Challenges for DRE Systems

The following challenges faced by distributed real-time and embedded (DRE) systems motivate our work on aspect frameworks for component-based model-integrated development:

1. Extra-functional constraints such as end-to-end timeliness must be satisfied while also ensuring the system's functional correctness.

2. Extra-functional constraints tend to cut across traditional endsystem and architectural layer boundaries.

3. Details of infrastructure mechanisms used by DRE systems have significant impacts on extra-functional properties and must be modeled in the overall analysis of system correctness.

4. The number and variety of system mechanisms that must be considered grows with the heterogeneity and scale of the system.

5. Due to all of the previous factors, analysis of correctness can be computationally expensive even for apparently simple applications.

Historically, many DRE systems have been developed manually and individually to assure their constraints will be met even under a set of worst case conditions known *a priori*. Unfortunately this has occurred at a very high cost both in engineering effort and in lost opportunities due to lengthy system development cycles. Furthermore, these static approaches are brittle with respect to the kinds of highly variable environments faced by the motivating applications described in Section 2: if the *a priori* assumptions are violated by the actual conditions the system encounters, then correctness cannot be assured.

1.2 A Generative Programming Approach

In recent years, significant emphasis has been placed on the design of configurable and customizable software. Such software is often built from modules combined to form frameworks addressing issues of interest to particular classes of applications. Each instance of a module may have its own unique customizations both between frameworks and when the same framework is refined for specific applications. Czarnecki and Eisenecker [10] describe this kind of development as being analogous to a luxury car assembly plant, in which each car might have its own customized fittings. Applying these ideas to middleware infrastructure

---

Sidebar 1: Aspect Oriented Design (AOD) vs Programming (AOP)

To manage complexity, readability, evolution and composition of a system, it is essential that proper attention be given to crosscutting concerns during the design stage itself even though the dominant decomposition of the system may be object-oriented. Analyzing a problem domain to decide which concerns are to be encapsulated as crosscutting *aspects* is the focus of AOD. Another item of focus for AOD is to make sure that the concerns thus identifed are complete and correct with respect to satisfying system requirements.

Once the different aspects have been identified, weaving them together is the focus of Aspect Oriented Programming. There are a variety of tools which support AOP, which are used by research projects - AspectC++, AspectJ, AspectWerkz, JAC. While strong AOP tools such as AspectJ [11] are available, their counterparts for languages such as C++ that are used for a majority of DRE system development, are yet to attain a sufficient level of maturity for our needs.

---

development is an emerging area of research. The confluence of QoS-aware component models [7, 8] with model-integrated computing [9] offers an important new paradigm for developing complex large-scale systems with stringent functional and extra-functional properties. However, many open issues such as configuration techniques for multiple infrastructure aspects must be addressed before this approach

will be widely applicable to real-world DRE systems.

In particular, while component technologies ease the packaging, assembly and deployment of *application components*, numerous constraints that cut across the application components and the supporting middleware infrastructure on which they run must still be configured manually or through manipulation by a higher-level modeling tool. In either case, gratuitous detail complicates the task, even though aspect-oriented modular structure is inherent in many of these configuration problems.

Furthermore, while QoS-aware component technologies provide *mechanisms* for configuring these crosscutting concerns, the configuration issues are often orthogonal to the particular component technology used. Configuring all details of a system, from the highest to lowest architectural levels, can make the implementation and extension of modeling tools unnecessarily complex. Clearly, exposing only certain key details to the component technology and higher level modeling tools while encapsulating the other details, is advantageous.

We therefore believe that such issues are best addressed just above the lowest common level from which both the application components and middleware mechanisms are built. Aspect-oriented design and Aspect-oriented programming [12] (see Sidebar 1) techniques deal with the separation of functional aspects of a system from the cross-cutting *aspects* and then compose these aspects and functional components to obtain system implementations. In this paper we describe how alternative Generative Programming techniques such as C++ Template Meta Programming [10] can be applied in the absence of suitable Aspect Oriented-Programming tools to configure aspects at this level. In particular, template meta-programming can be applied to configure ACE [13] primitives for both real-time application components and infrastructure mechanisms such as *reactors* [14] on which they are run, thus reducing complexity and increasing fidelity of the system.

This work enables adaptive and evolvable software systems in two main ways. First, it offers flexibility to customize system properties in response to constraints crosscutting the application and middleware levels. Second, it offers a rigorous and reusable common substrate for software development and customization, across both architectural layers and alternative component technologies.

1.3 Structure of this Paper

The rest of this paper is structured as follows. Section 2 describes a motivating real-world example that gives rise to the challenges described in Section 1.1. Section 3 discusses how problems with specification of system correctness involve details of both the application and its supporting infrastructure. We introduce logic for specifying system

constraints in Section 4. Section 5 describes our solution framework and explains in detail how our solution resolves the challenges faced by the example in Section 2. Section 6 examines related work and compares and contrasts our approach to other relevant approaches. Finally, Section 7 offers conclusions and describes future work.

2   Motivation

DRE applications such as integrated avionics mission computing systems [15] have benefited significantly from advances in middleware technology, and work is underway to apply QoS-aware component technologies to them as well [8]. However, these systems are relatively small-scale compared to next-generation DRE systems such as *autonomous agent* systems involving swarms of coordinated unmanned aerial vehicles (UAVs) [16] or teams of collaborating emergency rescue robots [17].

For autonomous agent systems, the need for individual systems (*i.e.* each UAV or robot) to communicate and coordinate their actions with one another means that the functional and QoS configuration of each single system must be adapted (potentially repeatedly) to reflect its interactions with other systems. For example, it is likely that each UAV would take to the air separately, but then once airborne would establish a formation with the others before proceeding to a specified destination. Each UAV would need to communicate with the others, which in a distributed middleware setting would involve sending messages between UAVs resulting in method upcalls on objects that actually perform services within the UAV application. Thus when a single system initiates a new interaction with another system, both systems experience a *mode change*. "Mode change" is a standard term used in the real-time literature to indicate a major change in the state of a system which results in a possibly different behavior of the system.

Ensuring safety and feasibility of operations in individual systems can be realized by (1) modeling a graph of method invocations, (2) decorating that graph with QoS attributes like execution times and rates of invocation, and (3) performing analysis over that graph. A key issue raised by this example is that the method invocation graph within each individual autonomous system must be augmented to reflect the interaction with the other autonomous systems. As we examine in greater detail in Section 3, this implies that the correctness of each system's individual QoS configuration does not necessarily imply correctness of the combined and interacting system-of-systems that results when teams of autonomous systems interact. Furthermore, because it may be intractable to consider all combinations of possible interactions *a priori*, run-time checking of new configurations may be necessary.

3   Evolving System Property Specifications

An abstract model of a system is used to analyze key properties like safety, liveness, and resource feasibility. Though the functional aspects of the system are represented in the model at an abstract level, we observe that it may also be necessary to model some of the key infrastructure mechanisms used in the implementation. These infrastructure mechanisms may have significant impacts on extra-functional properties and must be modeled in the overall analysis of system correctness.

Middleware typically offers different strategies to configure infrastructure mechanisms. Correct choice of strategies is crucial not only for the correct functioning of the infrastructure, but also is required to maintain safety and liveness properties. In this section, we present a simple, but sufficient example to illustrate the need to include infrastructure mechanisms during modeling of a system. Section 3.1 first describes the properties of a simple set of infrastructure aspects that illustrate problems of safety, timeliness, and schedulability. Section 3.2 then examines how these aspects influence the specification and checking of system correctness.

3.1   Middleware Infrastructure Aspects

CORBA [18] based ORBs are increasingly being used in distributed systems with real-time constraints. Implementation of an ORB [19] involves mechanisms like Reactors and Leader-Follower thread pools (see Sidebar 2). While modeling DRE systems, it becomes necessary to consider key infrastructure mechanisms like the ORB core reactor, the number of threads used to receive incoming GIOP [20] requests, and the topology of method invocations that generate outgoing GIOP requests. In this section, we describe one such strategy used to configure ORB core infrastructure – *Reply Wait Strategy* – to illustrate the importance of including this level of detail in a system model. In the course of this example, we find that the type of reply wait strategy chosen at one end-sytem affects the real-time characteristics as well as safety properties of other end-systems, hence reply wait strategy is an *aspect* which cuts across end-system boundaries.

3.1.1   ORB Reply Wait Strategies

In CORBA, when a client makes a remote two-way function call, the caller's thread needs to wait until it receives a reply back from the server before continuing to execute the calling method. This is in accordance with the semantics of a two-way function call. There are different strategies to wait for the reply, each having different implications for safety

and liveness. Two different strategies to wait for the reply are illustrated here:

- Wait on Connection
- Wait on Reactor

We use TAO [19] to illustrate the impact of these strategies on the safety and liveness properties of distributed applications. A simple representative example consists of a client communicating with a server using a two-way remote call and passing a callback object reference. The server makes a remote call back to the object corresponding to the reference passed from the client. This could repeat for a finite number of times, the back-and-forth calls then being stopped by some kind of a counter. In ORB literature, this kind of sequence of calls is termed "Nested Upcalls". Without loss of generality, we first assume that there is a single thread in the client and server.
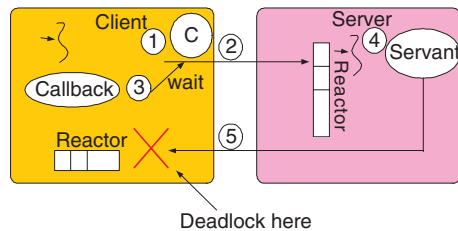

Figure 1: Waiting for the reply on the connection

Sidebar 2: Key Patterns in TAO

The architecture of TAO is based on the network programming patterns described in [14]. We outline three fundamental patterns used in TAO that are relevant to the discussion in this paper:

- **Reactor** is an event handling design pattern used in network programming to demultiplex events from multiple sources using just a single thread. This design pattern is used in ORBs to demultiplex and dispatch incoming requests and replies from peer ORBs. Event handlers like request and reply handlers are registered with a reactor. The reactor uses a synchronous event demultiplexer, *e.g.* the UNIX *select* system call, to wait for data to arrive from one or more ORBs. When data arrives, the synchronous event demultiplexer notifies the reactor, which then dispatches the appropriate registered event handler based on the event source.

- **Leader/Followers** is an architectural design pattern that provides an efficient concurrency model where multiple threads take turns detecting, demultiplexing, dispatching, and processing requests and replies from peer ORBs.

Wait on Connection: In this strategy, illustrated in Figure 1, the following sequence of events takes place within the ORB layer:

1. As soon as the client makes a remote call, the client ORB actively establishes a connection `C` to the server ORB.

2. The parameters to the remote call are marshalled by the client stub, a GIOP Request is formed and sent to the server using `C`.

3. The sole client thread waits for the reply on the connection `C` using a blocking `recv` call.

4. The request is received by the server and dispatched to the appropriate skeleton. The skeleton marshals the parameters and the *upcall* is made to the servant.

5. The servant implementation in this example uses the callback object reference (passed as parameter to the remote call) to make a remote call back to the client.

Since the sole thread on the client side is blocked on a system call waiting for a reply from the server, there is no thread to accept the incoming request. This results in a deadlock, where the client is waiting for a reply from the server and the server is blocked on the client for a reply. The situation can be improved by having a pool of threads listening for input requests using the Leader-Follower model (see Sidebar 2). But even with this model, when the number of outstanding requests exceed the number of threads, the ORB ceases to accept any more requests and this will result in a deadlock in the case of nested upcalls.

Wait on Reactor: In this strategy, the sequence of calls is the same as the previous strategy until the request is written to the connection stream. After that, instead of waiting on the connection for the reply, the caller thread waits on the ORB core reactor, which provides synchronous demultiplexing of I/O events. This demultiplexing allows incoming requests to be accepted while waiting for replies (see Sidebar 2). The (nested) callback request from the server is accepted and the call is completed eventually, thus avoiding deadlock (see Figure 2).

It should be noted that the upcall for the incoming request is made in the same thread context as that of the outgoing call. There could be multiple incoming requests before the reply for the initial outgoing call arrives. The processing of the reply for the initial outgoing call can be done only after processing of all the incoming requests, that arrived before its reply, is completed. This results in blocking delays in completion of outgoing remote calls. Section 4.3 describes this process in detail.

Observation: The above discussion illustrates that it is important to choose appropriate strategies at fine levels of detail in a middleware infrastructure. Depending on the nature of application properties, *e.g.* nested upcalls, this choice may drastically affect liveness properties as shown in the example. Therefore, such details need to be taken into consideration when doing analysis of the system model.
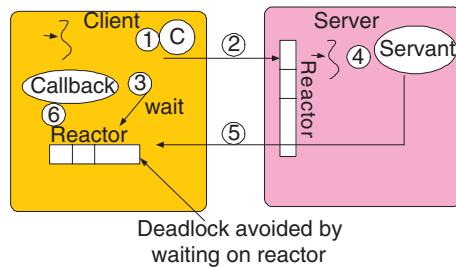
Figure 2: Waiting for the reply on the reactor

3.2 Specification of System Correctness

The example discussed in Section 3.1 illustrates some of the problems encountered when building systems out of various components. A component needs to be configured based on the application or execution environment in which the component is used. This is because the component might be designed to be used by multiple application enviornments, which is certainly true in the case of CORBA ORBs. The configuration can be based on system properties or constraints evaluated statically or dynamically and may need to be changed during the course of the application execution because of *mode changes* in the application. The correctness of the system needs to be maintained even across such mode changes. We outline some of the system properties which need to be maintained even under changing environments:

- Timeliness constraints
- Schedualability
- Safety properties

Timeliness Constraints: Even under changing environmental conditions, real time systems mandate tasks to be completed before their deadlines. Based on this, the infrastructure mechanisms might need to be reconfigured to adapt to the new environment. The new configuration could affect the system properties in an adverse way and hence this should be taken into account during the system modelling phase.

In the example above, blocking factors are simpler to calculate if the reply wait strategy is configured as *WaitOn-Connection* since the thread waits only till the reply is received and no incoming requests are processed in between. If we do an analysis of the system model based on this and later change the configuration to *WaitOnReactor*, then the analysis that we did would not be valid anymore since the blocking factor need to be considered for analysing the new configuration. This could result in violation of timing requirements.

Schedulability: As illustrated by the example in Section 3.1, nested upcalls could affect the schedulability of a system. Such nested upcalls introduce blocking times [21], which need to be accounted for while doing schedulability

analysis like RMA [22]. There is a possibility that the system might be under-utilized because of considering blocking factor when in reality the configuration of the infrastructure does not allow blocking to happen. This is true when we configure the infrastructure with a reply wait strategy of *WaitOnConnection*. Both runtime and static admission control policies should take this into account when making schedulability decisions.

Safety Properties: Changing system environments could affect the safety properties of the system. For example, interaction between two independently developed components could result in the deadlock illustrated in Section 3.1. Endsystems might be configured with *WaitOnConnection* and runtime mode changes might then cause the application to change to a state where there are nested upcalls between components. In this scenario, the reconfiguration of the system should include doing the appropriate call graph analysis and make sure that the safety properties of the system are not violated. If the callgraph after a mode change detects a nested upcall, the infrastructure could be reconfigured with *WaitOnReactor* reply wait strategy, if necessary.

To analyze the above properties, the first step is formalize the constraints in terms of a logic for what constitutes a "correct" and "safe" system. We need to be able to formally define what a deadlock means and also derive rules which enable us to detect the possibility of a deadlock based on system properties. We introduce a first order logic called the *Infrastructure Configuration Logic* to formalize and verify some of the system properties based on the configuration used in the infrastructure mechanisms.

4 Infrastructure Configuration Logic

Our logic is designed to allow description of scenarios that occur in systems using CORBA-like infrastructure mechanisms. Though we apply this logic using TAO as an example, it is easy to generalize this logic to fit other infrastructure mechanisms.

4.1 Logic Notations

For any remote function call in CORBA, there is a source, usually called the *client* and a destination, usually referred to as the *server*. The variables f, f_1, etc. are used to denote functions. It should be noted that two different variables can be assigned the same function value. For example, f_1 and f_2 could both have the value *foo*, which is a function that resides in a remote CORBA object. To indicate that a function f_1 calls a remote function function f_2, we say

$$f_1 \rightsquigarrow f_2$$

To illustrate a call-chain involving three different functions, $f_1 \leadsto f_2 \leadsto f_3$ indicates that f_1 makes a remote call to f_2 which in turn makes another remote call to f_3.

Remote functions are embedded in CORBA objects which in turn are activated in server processes. The variables P, P_1, etc. range over processes which host CORBA objects.[1] The relation *HostedIn* is used to denote the process in which a CORBA object, and hence the remote object method or function, is hosted. *HostedIn(f, P_1)* indicates that the function f is hosted in process P_1. Finally, we use the transitive closure $\leadsto^+$ to indicate that a function calls another function indirectly as part of a call-chain. In the above example, we can assert that $f_1 \leadsto^+ f_3$. We denote a call chain by variables C, C_1, etc. .

Each process hosting CORBA objects may configure the ORB that it uses, with an appropriate strategy to wait for the reply. The two strategies described in Section 3.1 are denoted using *WaitOnConnection* and *WaitOnReactor*. The relation *ReplyWaitStrategy* is used to describe the reply wait strategy used in a particular process. It can also be expressed as a predicate which evaluates to either true or false. For the discussion in this paper, we consider this relation as a primitive construct whose semantics is illustrated in the discussions in Section 3. As part of our future work, we plan to continue extending appropriate formalisms to define such constructs in greater detail.

We introduce two more *operators* in our logic called *ThreadCount* and *BlockingTime*. *ThreadCount(P)* indicates the number of threads configured in the ORB to listen to incoming requests. Different threads take turns listening according to the Leader-Follower pattern (see Sidebar 2). *BlockingTime(f)* indicates the time a function call will take before execution continues at the point after the function call, in the calling method.

4.2 Safety Properties

As explained in Section 3.1, a call chain could end up in a loop resulting in a nested upcall. This could result in a deadlock based on the configuration of the wait strategy for the ORB infrastructure. We introduce the operator *Deadlock* which indicates the possibility of a deadlock happening on a given call-chain.

Assuming that C is a call-chain $f_1 \leadsto^+ f_n$ representing

$$f_1 \leadsto \cdots f_i \leadsto \cdots f_n$$

possibility of a deadlock in this call chain can be verified by

$$Deadlock(C) \stackrel{\text{def}}{=} \exists P_j, \exists f_i \mid$$
$$\|\{f_i \mid HostedIn(f_i, P_j)\}\|$$
$$> ThreadCount(P_j)$$
$$\wedge \quad ReplyWaitStrategy(P_j, WaitOnConnection)$$

The term $\|\{f_i \mid HostedIn(f_i, P_j)\}\|$ represents the number of functions in the call-chain that are hosted in P_j. According to the above logic, if there are more than *ThreadCount(P_j)* functions hosted in P_j that are part of the same call-chain, then there is a possibility of deadlock if the ORB is configured with a reply wait strategy of *WaitOnConnection*. This is because the all threads will be exhausted waiting for replies and no threads left to accept incoming requests. To avoid this possible deadlock, the infrastructure must be configured with a reply wait strategy of *WaitOnReactor* instead of *WaitOnConnection*.

4.3 Schedulability

When the *WaitOnReactor* strategy is used to wait for replies, incoming requests can be processed while a reply for an already issued request is outstanding. To determine whether the system is schedulable or not in the presence of such interleaved calls, the increased complexity of the blocking time for tasks should be taken into account. We illustrate
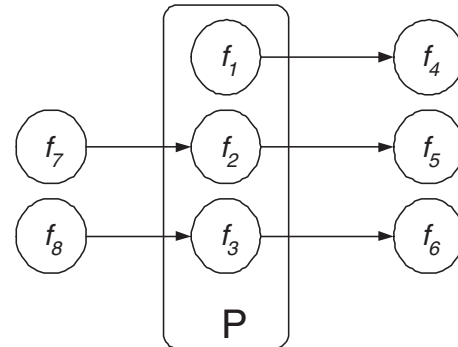

Figure 3: Example scenario to illustrate Blocking factor

this with an example as shown in Figure 3. There are 8 functions. The three functions f_1, f_2, f_3 are hosted in the same process P and they make remote calls to functions f_4, f_5 and f_6 respectively. f_2 and f_3 are invoked as a result of remote calls from f_7 and f_8. For the purposes of simplicity we consider only direct remote calls, although it is easy to extend this to a remote call chain using the transitive closure property.

Given the above scenario, Figure 4 shows the thread of control flowing through the different components. Note that the stubs and skeletons are only shown for completeness and do not significantly impact this analysis. We ignore the actions before function f_1 starts execution. f_1 makes a remote call to f_4. The flow of control passes through the stub code

for f_4 and eventually blocks on the reactor waiting for the reply from the server hosting f_4.

Blocking on the reactor enables processing of incoming requests even in the presence of outstanding replies. In our example, a request for f_2 comes in when the ORB is waiting for reply from f_4. The thread that was blocked on the reactor makes the upcall to f_2. f_2 now makes another remote call to f_5. Again, the ORB waits for the reply from its peer. Meanwhile, the reply from f_4 arrives. We cannot unwind the thread stack at this point since that would break the two-way semantics of f_2. So the reply from f_4 has to be queued until the reply from f_5 has been processed and f_2 has finished execution. This causes an unnecessary blocking delay for f_1 since its reply has already arrived but cannot be processed.

We use the notation $f_1 \dashv f_2$ to indicate that f_1 is blocked by f_2. Note that this does not necessarily imply that $f_1 \rightsquigarrow^+ f_2$. In the example discussed above, there is no call-chain in which both f_1 and f_2 are involved, but still the relation $f_1 \dashv f_2$ holds. We also introduce the notion of a *list* used to represent a sequence of items. To talk about lists of arbitrary length, we use the binary functional operator "." in infix form. In particular, a term of the form $\tau_1.\tau_2$ designates a sequence in which τ_1 is the first element and τ_2 is the rest of the list. This is very similar to the CAR and CDR operators in LISP.
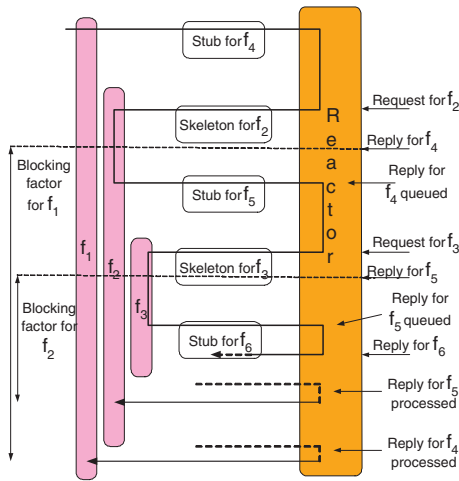

Figure 4: Flow of control with WaitOnReactor Strategy.

We now introduce a logic for evaluating the blocking time for a function. If there is a sequence of functions

$$[f_1, \ldots, f_i, \ldots, f_n]$$

such that

$$f_1 \dashv f_2 \dashv \ldots f_i \dashv \ldots f_n$$

then the blocking time for f_1 can be written as

$$BlockingTime(f) \stackrel{\text{def}}{=} ExecutionTime(f.tail)$$
$$ExecutionTime(f) = Execution\ time\ of\ f\ without\ any\ blocking$$
$$ExecutionTime(list) = \sum ExecutionTime(f_i),\ f_i \in list$$

Once the blocking time is calculated this needs to be taken into account while doing schedulability analysis using techniques like RMA with blocking factor [21].

5 Logic Implementation Schemes

In this section, we propose possible mechanisms for implementing the logic discussed in Section 4. Some of the assertions can be made at compile time and some others can be done only at run time. For example, a call graph detailing the function calls can be constructed at compile-time and analyzed for possible nested upcalls and deadlocks, but when an application makes a mode transition while running, it may not be possible to predict, *a priori*, the resulting system properties. In such cases we might have to resort to dynamic logic evaluation.

5.1 Static Analysis

Certain aspects of a system render themselves to be pre-configured at design time based on facts available at design time. A static analysis would suffice in such cases. C++ Template meta-programming [10] provides excellent mechanisms to do compile-time computations. This power combined with its applicability in writing configuration generators can prove to a valuable combination for implementing a logic analyzer. Facts in the logic can be asserted at compile-time and the results inferred can be used to generate appropriate configurations using generators. For example, using the logic discussed in Section 4.2, the existence of a deadlock can be asserted at compile-time, if the call-chain in the system can be determined *a priori*. The challenge here is to choose the appropriate template meta-programming constructs to represent the logic as well as the call graph.

C++ templates provide mechanisms to prohibit certain template instantiations [10]. If we could prohibit the instantiation of a template for certain combinations of system properties, that would serve as a compile-time checker for *invalid* combinations. One simple way to do this is to use template specialization. In effect, this is a way to make aspects type-safe in the absence of AOP tools. We show a glimpse of a configuration validator using template metaprogramming mechanisms in Sidebar 3.

Specialized template definitions can be provided for all invalid parameter combinations and prohibit instantiation of these by defining the template classes as having a private constructor or something similar. This would give a

compile-time error when the invalid set of system properties is used to instantiate the template. Alternatively, if the space of *valid* configurations is smaller, we could prevent instantiation of a base template, but allow instantiations of specializations representing the acceptable combinations.

Sidebar 3: Configuration Validator

```
//A simple example to illustrate compile-time
//verification of infrastructure configuration.
//IF meta-control structure adopted from
//template metaprogramming by Czarnecki[10]
template<bool Condition,
        typename TruePart,
        typename FalsePart>
struct IF
{
  typedef TruePart RET;
};
template<typename TruePart,
        typename FalsePart>
struct IF<false,TruePart,FalsePart>
{
  typedef FalsePart RET;
};
enum WaitPolicyT {WaitOnReactor,
                WaitOnConnection};
//Do we need bounded Blocking factor?
enum BFBoundT {BFBounded,
              BFUnBounded};

class InvalidCombination
{
  //Instantiation of this class gives compile error
private:
  ~InvalidCombination();
};

class ValidCombination
{
public:
  ~ValidCombination()
};

//Compile-time verifier to determine
//validity of choice of policies -
//For example, one cannot get a bounded
//blocking factor with a WaitOnReactor
//wait strategy
template<WaitPolicyT WaitPolicy,
        BFBoundT BFBound>
struct PolicyValidator
{
  typedef IF <WaitPolicy==WaitOnReactor,
              IF <BFBound==BFBounded,
                  InvalidCombination,
                  ValidCombination>::RET,
              ValidCombination
              >::RET RET;
};

//This one is a valid combination.
PolicyValidator<WaitOnReactor,
    BFUnBounded>::RET validator1;

//This one is an invalid combination.
//Gives compile error
PolicyValidator<WaitOnReactor,
    BFBounded>::RET validator2;
```

5.2 Dynamic Analysis

DRE systems undergo mode changes which may change the system behavior. These changes would require new set of configurations at the infrastructure level. Such configuration cannot be determined at static time and hence template meta-programming cannot be of any use here.

Under such situations an adaptive approach involving dynamic logic evaluation is necessary. This is especially relevant in applications requiring admission control. Languages like Prolog can be used to represent and evaluate the logic in some situations, or in our case we could build a simple expression structure and evaluator for use in C++. It should be noted that the run time evaluation of logic rules can be computationally complex and canonical forms like Horn clauses might be useful in reducing the complexity of computation within bounds.

5.3 Hybrid Analysis

A combination of the above two approaches is useful, if some system properties can be known *a priori* and some others would be known only at runtime. This approach balances on-line computation cost with flexibility by pre-compiling particular parameters of the on-line specification mechanisms for performance and predictability. One of the key question for the static part is whether we pre-compile too much and thus over constrain so that the solution becomes brittle in some environments, or too little so that computational complexity of on-line specification exceeds constraints.

6 Related Work

This work intersects with prior work in the following areas:

- Reasoning in concurrent and component-based systems
- Configuration of Component based systems
- Model-based systems integration

Logic applied to Hierarchical Scheduling: Task/Scheduler Logic (TSL) [23, 24] has been used to reason about concurrency in component based software systems. Each component might come under the purview of one of a hierarchy of schedulers, each imposing its own set of restrictions on the type of resources that can be used. TSL uses first order logic to represent tasks, resources, locks and schedulers. Such reasoning is essential in component based systems to make more efficient uses of resources. Components are executed in environments which may be different from the environments that they are developed in. TSL can be used to find errors in system

code, for example, using a lock in a component which will eventually be run as an interrupt handler. There are different kinds of locks like regular mutex locks, recursive locks, readers-writer lock, etc. Based on the environment and the call graph of functions, TSL can be used to infer the type of lock to be used by a particular component under a particular context.

RMA using C++ template-metaprogramming: C++ Template metaprogramming provides powerful mechanisms to do compile-time computations. Veldhuizen [25] shows an example of factorial computation at compile-time. Since RMA involves static schedulability analysis of a set of tasks and the periods of these tasks are known *a priori*, a static analysis can be done using template metaprogramming constructs. If a set of tasks are not guaranteed to be scheduled according to the RM utilization bound, a compile time error will be generated. Deters [26] describes an implementation of Rate-Monotonic Analysis (RMA) within the C++ parametric type system that provides C++ real-time software developers a good way to reason with types at the source level about recurrent tasks and deadlines. Using this approach, a program can be considered incorrect, raising type errors at compile time, if a given set of tasks is not statically schedulable. Similarly, this compile-time "metaprogram" can adjust a task set so as to become feasible and this this analysis is performed inside the C++ type system, which allows a very natural integration into C++ programs.

Bossa: Bossa [27] is a special-purpose language dedicated to the development of scheduling policies. By providing a high-level abstraction, developing scheduling modules become less error-prone. This approach also provides a clean separation of the scheduling policy from the actual mechanisms. Moreover, dealing with high level abstractions makes possible the verification of important safety properties that are specific to the domain of scheduling.

CIAO: Component Integrated ACE ORB (CIAO) [7] is a QoS-aware open source implementation of the CORBA Component Model (CCM) [28] specification. CIAO currently aims to provide component-oriented paradigm to the distributed, real-time, embedded (DRE) system developers by abstracting DRE-critical systemic aspects, such as QoS requirements, RT policies, as installable/configurable units supported by the component framework. Promoting these DRE-critical aspects as first-class meta data disentangles code for controlling these non-function aspects from application logic and makes DRE system development more flexible. Since mechanisms to support various DRE-critical non-functional aspects can be easily verified, CIAO will also make configuring and managing these aspects easier.

Reasoning in CCM: Cadena [29] is an integrated GUI environment for building and modeling Corba Component

Model (CCM) [28] systems. Its philosophy is based on the fact that reasoning about correctness properties is essential in component-based designs. CCM architecture defines different roles during the lifecycle of a CORBA component. Configuring a component is done through XML based descriptors which are tedious to write manually. Cadena provides a component assembly framework supporting a variety of visualization and programming tools for developing component connections. It provides model checking for verifying correctness properties of CCM systems derived from CCM IDL and XML. It does this based on specifications of a component along with component assembly information combined with Cadena specifications. It also provides facilities for defining component types, specifying dependency information and transition system semantics for these types.

Model Integrated Computing(MIC): Integration of embedded systems using different components require a great deal of *a priori* modeling and analysis. The key element in MIC is that it extends the scope and usage of models such that they form the "backbone" of a model-integrated system development process. The Generic Modeling Environment [30, 31] is a configurable toolkit for creating domain-specific modeling and program synthesis environments. The generated domain-specific environment is then used to build domain models that are stored in a model database. These models are used to automatically generate the applications or to synthesize input to different COTS analysis tools.

7 Conclusions and Future Work

This paper highlights the importance of including aspects that cut across infrastructure mechanisms and application descriptors when reasoning about the correctness of a DRE system. We presented an infrastructure configuration logic that offers a way to check safety properties and schedulability of DRE systems. From the results of evaluating this logic we can infer the appropriate strategy to use at particular places in the supporting middleware infrastructure, or offer proof why no configuration is acceptable, *e.g.* , for purposes of admission control.

We presented an example illustrating the application of this logic to the TAO ORB core infrastructure. We examined solutions that can be employed to configure the infrastructure mechanisms based on the result of evaluating the logic, of which C++ static template metaprogramming is a very powerful one for static configurations. For dynamic adaption, hybrid solutions are necessary involving partially preconfigured mechanisms that are reconfigured at runtime.

8 Acknowledgement

References

[1] Object Management Group, *CORBA Components*, OMG Document formal/2002-06-65 ed., June 2002.

[2] D. Alur, J. Crupi, and D. Malks, *Core J2EE Patterns: Best Practices and Design Strategies*. Prentice Hall, 2001.

[3] Microsoft Corporation, "Microsoft .NET Development." msdn.microsoft.com/net/, 2002.

[4] Object Management Group, *The Common Object Request Broker: Architecture and Specification*, 2.6.1 ed., May 2002.

[5] W. A. Domain, "Extensible Markup Language (XML)." `http://www.w3c.org/XML`.

[6] J. Snell and K. MacLeod, *Programming Web Applications with SOAP*. O'Reilly, 2001.

[7] N. Wang, D. C. Schmidt, A. Gokhale, C. D. Gill, B. Natarajan, C. Rodrigues, J. P. Loyall, and R. E. Schantz, "Total Quality of Service Provisioning in Middleware and Applications," *The Journal of Microprocessors and Microsystems*, vol. 27, pp. 45–54, mar 2003.

[8] N. Wang and C. Gill, "Improving Real-Time System Configuration via a QoS-aware CORBA Component Model," in *submitted to the Hawaii International Conference on System Sciences, Software Technology Track, Distributed Object and Component-based Software Systems Minitrack, HICSS 2003*, (Honolulu, HW), HICSS, Jan. 2003.

[9] A. Ledeczi, A. Bakay, M. Maroti, P. Volgysei, G. Nordstrom, J. Sprinkle, and G. Karsai, "Composing Domain-Specific Design Environments," *IEEE Computer*, Nov. 2001.

[10] K. Czarnecki and U. Eisenecker, *Generative Programming: Methods, Tools, and Applications*. Boston: Addison-Wesley, 2000.

[11] The AspectJ Organization, "Aspect-Oriented Programming for Java." `www.aspectj.org`, 2001.

[12] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin, "Aspect-Oriented Programming," in *Proceedings of the 11th European Conference on Object-Oriented Programming*, June 1997.

[13] Center for Distributed Object Computing, "The ADAPTIVE Communication Environment (ACE)." www.cs.wustl.edu/~schmidt/ACE.html, Washington University.

[14] D. C. Schmidt, M. Stal, H. Rohnert, and F. Buschmann, *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects, Volume 2*. New York: Wiley & Sons, 2000.

[15] B. S. Doerr and D. C. Sharp, "Freeing Product Line Architectures from Execution Dependencies," in *Proceedings of the 11th Annual Software Technology Conference*, Apr. 1999.

[16] US Navy Program Executive Office (Cruise Missiles and Joint Unmanned Aerial Vehicles), ""Unmanned air vehicle makes successful shipboard landing"." www.mediacen.navy.mil/pubs/allhands/mar00/pg6g.htm, 2000.

[17] E. R. Z. Zhu, K. Rajasekar and A. Hanson, "Panoramic Virtual Stereo Vision of Cooperative Mobile Robots for Localizing 3D Moving Objects," in *Proceedings of the IEEE Workshop on Omnidirectional Vision (OMNIVIS'00)*, IEEE, 2000.

[18] Object Management Group, *Real-Time CORBA Specification*, 1.1 ed., Aug. 2002.

[19] Center for Distributed Object Computing, "The ACE ORB (TAO)." www.cs.wustl.edu/~schmidt/TAO.html, Washington University.

[20] G. Coulson and S. Baichoo, "Implementing the CORBA GIOP in a High-Performance Object Request Broker Environment," *ACM Distributed Computing Journal*, vol. 14, Apr. 2001.

[21] G. C. Buttazzo, *Hard Real-Time Computing Systems*. Norwell, Massachusetts: Kluwer Academic Publishers, 1997.

[22] M. H. Klein, T. Ralya, B. Pollak, R. Obenza, and M. G. Harbour, *A Practitioner's Handbook for Real-Time Analysis: Guide to Rate Monotonic Analysis for Real-Time Systems*. Norwell, Massachusetts: Kluwer Academic Publishers, 1993.

[23] A. Reid and J. Regehr, "Task/Scheduler Logic: Reasoning about Concurrency in Component-Based Systems Software." `www.http://www.cs.utah.edu/~regehr/papers/tsl/tsl-pdf.pdf`, 2002.

[24] J. Regehr, A. Reid, K. Webb, and J. Lepreau, "Composable Execution Environments." `http://www.cs.utah.edu/flux/papers/cee-flux-tn-02-02/`, 2002.

[25] T. Veldhuizen, "Using C++ template metaprograms," *C++ Report*, vol. 7, May 1995.

[26] M. Deters, C. Gill, and R. Cytron, "Rate-Monotonic Analysis in the C++ Typesystem," in *Proceedings of the 9th Real-time/Embedded Technology and Applications Symposium (RTAS)*, (Washington, DC), IEEE, May 2003.

[27] L. P. Barreto and G. Muller, "Bossa: a language-based approach to the design of real-time schedulers," in *10th International Conference on Real-Time Systems (RTS'2002)*, (Paris, France), Mar. 2002.

[28] N. Wang, D. C. Schmidt, and C. O'Ryan, "An Overview of the CORBA Component Model," in *Component-Based Software Engineering* (G. Heineman and B. Councill, eds.), Reading, Massachusetts: Addison-Wesley, 2000.

[29] J. Hatcliff, W. Deng, M. Dwyer, G. Jung, and V. Prasad, "Cadena: An Integrated Development, Analysis, and Verification Environment for Component-based Systems," in *Proceedings of the International Conference on Software Engineering*, (Portland, OR), May 2003.

[30] G. Karsai, S. Neema, A. Bakay, A. Ledeczi, F. Shi, and A. Gokhale, "A Model-based Front-end to ACE/TAO: The Embedded System Modeling Language," in *Proceedings of the Second Annual TAO Workshop*, (Arlington, VA), July 2002.

[31] S. Neema, T. Bapty, J. Gray, and A. Gokhale, "Generators for Synthesis of QoS Adaptation in Distributed Real-Time Embedded Systems," in *Proceedings of the ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering (GPCE'02)*, (Pittsburgh, PA), Oct. 2002.