Configuration Management for Highly-Customizable Software *

Matti A. Hiltunen
Department of Computer Science
University of Arizona
Tucson, AZ 85721, USA
hiltunen@cs.arizona.edu

September 30, 1998

Abstract

Customizable operating systems, database systems, and communication subsystems have demonstrated many advantages of customization, including considerable performance improvements. One common approach for constructing customizable software is to implement it as a collection of modules that can be configured in different combinations to provide customized variants of the software. Typically, ad hoc methods are used to determine which modules may be combined. Such methods require intimate knowledge of the modules and their interactions or the configuration will not behave as expected. In this paper, we present a methodology that simplifies the difficult task of constructing correct custom variants of highly-customizable software. The methodology is based on identifying relations between software modules that dictate which combinations are correct. We also introduce a configuration support tool that, based on these relations, allows only correct configurations to be created.

1 Introduction

The recent research on customizable software components such as operating systems [1, 2, 3, 4], file systems [5, 6, 7], database systems [8, 9, 10], and communication subsystems [11, 12, 13, 14, 15] has demonstrated many advantages of customization. For example, it allows the implementation of a software component to be optimized for the requirements of its users as well as for the characteristics of the execution environment. As a result, the performance may be considerably better than that of a traditional non-customizable software component since no unnecessary features will be implemented and the algorithms used can be optimized for the environment. The process of customizing a software component can be difficult, however, and may easily result in versions that do not operate correctly in all situations. We call this the problem of *configuration management*, i.e., the process of managing the task of constructing correct custom versions of highly-customizable software components.

We have developed an approach to constructing highly-customizable middleware services. In our approach, a customizable service is implemented as a collection of modules called *micro-protocols*. A custom instance of the service is then constructed by combining a chosen subset of the micro-protocols. The distinctive feature of our approach is the *two-level composition model* where a system is constructed

*This work supported in part by the Office of Naval Research under grants N00014-91-J-1015, N00014-94-1-0015, and N00014-96-0207 and the Defense Advanced Research Projects Agency under grant N66001-97-C-8518.

hierarchically of protocols, each of which implements a well-defined service, and each protocol is composed internally of micro-protocols, each of which implements a well-defined abstract service property. In this model, the interactions between protocols are restricted to the operations defined in each protocol's service interface, whereas micro-protocols interact using shared variables and a flexible event mechanism. An example of a service implemented by a protocol might be atomic multicast, while properties implemented by micro-protocols could include consistent message ordering (fifo, causal, total) and delivery atomicity.

Configuration management is an important issue in our micro-protocol approach since typically micro-protocols cannot be combined arbitrarily. For example, micro-protocol m_1 may require that micro-protocol m_2 be in the configuration to operate correctly. Furthermore, some micro-protocols may not have been designed to be used together and thus combining them may result in a configuration that operates incorrectly. We say a configuration is *correct* when all such configuration constraints between micro-protocols are satisfied. Understanding and keeping track of such constraints can become difficult if the configurable service has a large number of micro-protocols. Thus, the challenge is to develop an approach for service configuration that allows users of the service to create correct configurations without having intimate knowledge of the micro-protocols and the configuration constraints between them.

The contribution of this paper is a new approach to the configuration management for highly-customizable software. This approach is founded on understanding the fundamental relations between service properties and between software modules that make up the configurable service. We first identify the relations between abstract service properties that dictate which combinations of properties are legitimate and then identify relations between the micro-protocol that dictate which combinations are correct. These relations are utilized in a configuration support tool that allows a service user to configure only correct versions. Although our configuration management methodology is introduced in the context of the micro-protocol approach, the relations also apply to software modules in other types of modular configurable systems. As a result, the methods and tools developed here could be applied to many other systems as well.

2 Properties of a service

The design of a highly-configurable service starts from identifying the aspects of the service that we want to make customizable. We call these aspects the properties of the service. When the properties of interest have been identified, we have to determine which property combinations are feasible.

2.1 Example: Properties of group RPC

Consider the abstract properties of a configurable group RPC service; group RPC service is a variant of RPC service where the client request is multicast to a server group rather than a single server. An RPC call can be *blocking* (synchronous), i.e., the client thread is blocked until the response arrives, or *non-blocking* (asynchronous), i.e., the client thread returns immediately. Different orders can be specified for the execution of concurrent calls on the servers. For example, *fifo order* guarantees that all calls issued by any one client are executed in the same order by all group members, while *total order* guarantees that all calls are executed

in the same order on all servers. Furthermore, an RPC with the *reliable communication* property tolerates transient communication failures, in particular, we assume it guarantees that if one server receives a client call, all servers will eventually receive the call. The *bounded termination* property guarantees that a call always terminates within a specified time. If the server has not responded by this deadline, the call returns with an indication of failure. Finally, different properties can be defined relative to how *orphans*—that is, server computations associated with clients that have failed—are handled. Options for dealing with orphans include *interference avoidance*, where any new calls from the recovered client are not processed until the orphans finish their execution, and *orphan termination*, where orphans are terminated upon detection.

A number of group RPC properties can be defined related to what guarantees are given concerning the execution of the client call at the server, both when the call returns successfully and when the call returns unsuccessfully. The *unique execution* property states that the client call is not executed more than once and the *atomic execution* property states that the call is either executed completely or not at all, i.e., never partially. With multiple servers, we must consider the case where the call is successfully executed only on some of the servers. A range of *acceptance* properties can be specified based on how many servers must succeed for RPC to be considered successful. Possibilities range from successful execution at only one server to successful execution at all servers. Finally, a variety of *collation* properties can be defined to specify how to handle multiple replies. The possibilities include returning any one reply, all replies, or the result of a function that maps all replies into one result. Further details on the properties of group RPC service, as well as of other services, can be found in [16].

2.2 Relations between properties

It turns out that any combination of abstract service properties is not feasible because all properties are not independent—for example, there may be two properties p_i and p_j such that there is no way to implement the service so that p_i is guaranteed but p_j is not guaranteed. These types of constraints can be stated as relations between properties. We define the relations in terms of satisfying a property in a system execution as follows. Let s be a system, e_s be an execution of s, and *SYS* be the set of all systems. Let $sat(e_s, p_i)$ be true if property p_i is satisfied for execution e_s and false otherwise. We identify three basic relations:

- *Conflict:* $con(p_i, p_j) \Leftrightarrow \forall\ s \in SYS\ \exists\ e_s : \neg sat(e_s, p_i) \vee \neg sat(e_s, p_j)$

  That is, two properties conflict when no system can guarantee both p_i and p_j for all executions.

- *Dependency:* $dep(p_i, p_j) \Leftrightarrow \forall\ s \in SYS\ \forall\ e_s : sat(e_s, p_i) \Rightarrow sat(e_s, p_j)$

  That is, one property p_i depends on another p_j when satisfying p_i requires that p_j also be satisfied; in other words, there is no way to satisfy p_i without p_j.

- *Independence*: $ind(p_i, p_j) \Leftrightarrow \neg con(p_i, p_j) \wedge \neg dep(p_i, p_j) \wedge \neg dep(p_j, p_i)$

  That is, two properties are independent if they do not conflict and neither one depends on the other.

These relations determine, in essence, the set of feasible combinations, $\mathcal{C}$, of any two properties p_i and p_j:

- $con(p_i, p_j)$: $\mathcal{C} = \{p_i, p_j\}$.
- $dep(p_i, p_j)$: $\mathcal{C} = \{p_i \wedge p_j, p_j\}$.
- $ind(p_i, p_j)$: $\mathcal{C} = \{p_i, p_j, p_i \wedge p_j\}$.

Naturally these rules generalize to any number of properties.

Consider the relations between properties of group RPC as examples. A conflict relation can be identified between the properties of blocking and non-blocking call since an RPC call naturally cannot be both blocking and non-blocking at the same time. A dependency relation can be identified between total order and reliable communication since for total order to be satisfied all servers must process the same set of calls in the same order and thus reliable communication property must be satisfied. Finally, total order and blocking (or non-blocking) are independent since a service can be blocking without total order, have total order without being blocking, and have total order and be blocking.

2.3 Dependency graphs

Dependency graphs are a graphical method for expressing the preceding relations between properties. A dependency graph is a directed, not necessarily acyclic, graph where each basic node represents a property and each edge a dependency. In addition, unlabeled *choice nodes* that encapsulate two or more nodes are provided to represent choice of properties resulting from conflicts. Specifically, the relations are graphically represented as follows:

- $dep(p_i, p_j)$: an edge from p_i to p_j.
- $con(p_i, p_j)$: p_i and p_j are included in a choice node.
- $ind(p_i, p_j)$: p_i and p_j are not included in the same choice node, and no path connects p_i and p_j.

Dependency graphs are simplified by omitting transitive dependencies. For example, if p_i depends on p_j and p_j depends on p_k, then the transitive edge from p_i to p_k is omitted.

Figure 1 shows the dependency graph of the group RPC properties discussed in section 2.1. Choice nodes in the graph are represented by shaded boxes that may include two or more property nodes. Note that *collation* and *acceptance* are represented as choice nodes since they represent sets of different collation and acceptance properties, respectively. Furthermore, the figure includes an additional *basic RPC* node that represents a simple RPC with no ordering, reliability, or other properties. The basic RPC indicates the minimal combinations of properties. In particular, any group RPC must be either blocking or non-blocking and must have some acceptance and some collation property.

Some of the relations in the figure have already been discussed in this section and due to space constraints we cannot provide details on all. However, we highlight a few interesting examples. The graph indicates that total order and fifo order are independent. The reason is that the definition of total order does not state that the order in which each client sends its calls (fifo) is to be maintained in the total order. This means

that three combinations of ordering guarantees are possible for a group RPC service: fifo, non-fifo total, and fifo total. The figure also indicates that orphan termination depends on interference avoidance. This follows from the fact that if orphans are terminated as quickly as they are detected, there will be no interference, i.e., interference avoidance property is satisfied. Finally, atomic execution is independent of orphan termination because atomic execution only requires all or nothing property and if a call is terminated as an orphan, atomicity can be satisfied by restoring the state of the server to the one before the execution of the call was started.

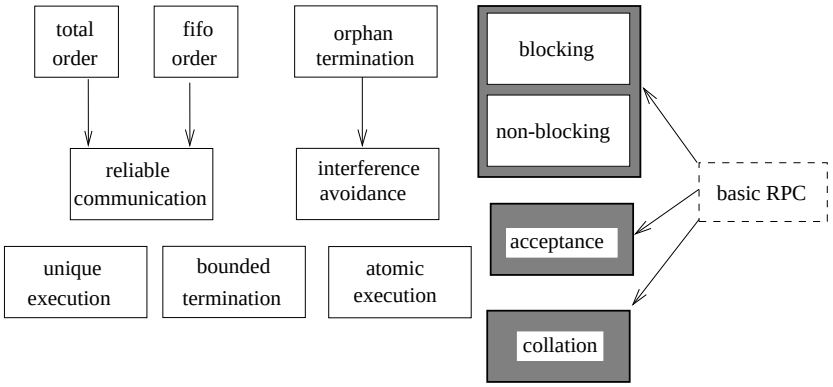

Figure 1: Dependency Graph of a Group RPC Service

A dependency graph represents all possible legitimate combinations of service properties in any implementation, and thus, it sets an upper bound on the customizability of any configurable service implementation. In the case of group RPC service, the number of semantically different legitimate service variants is 240 even without considering the numerous possible collation and acceptance policies. Obviously, for a graph with 10 properties (excluding acceptance and collation), the maximum number of configurations would be $2^{10} = 1024$ if all properties were independent.

3 Configurable modules

The service properties and the dependency graph provide the foundation for the implementation of a configurable service. Actual implementation may still be difficult, however, and design decisions may affect the configurability and performance of the resulting service. In this section, we outline the micro-protocol approach and, in a manner analogous to properties, discuss which combinations of micro-protocols provide a correct service variant.

3.1 Micro-protocol approach

Our implementation approach is based on a two-level view of system composition as illustrated in figure 2: a system is constructed of services, with each service being composed of modules that implement the abstract properties of the service. Services can typically be combined using traditional hierarchical

methods, such as supported by the *x*-kernel [11], but the traditional methods are often inadequate for the more closely interrelated modules that implement service properties. A detailed comparison of the micro-protocol approach to other approaches is beyond the scope of this paper but details can be found in [15]. In our approach, each service is implemented as a *composite protocol* that is constructed of software modules referred to as micro-protocols. Micro-protocols in a composite protocol interact using flexible event mechanisms as well as shared data. The result is a two-level model that supports flexible interaction and data sharing between modules when necessary, but also allows the strict hierarchical separation and proscribed interaction through a uniform protocol interface found in current hierarchical systems. One practical implication of the two-level model is that services are relatively independent from one another and thus can be developed in isolation, whereas micro-protocols in a composite protocol are tightly interconnected and would typically be developed together. However, if the events and shared data structures in a composite protocol are well designed and documented, it would be feasible for someone else to develop and add new micro-protocols to an existing service implementation.

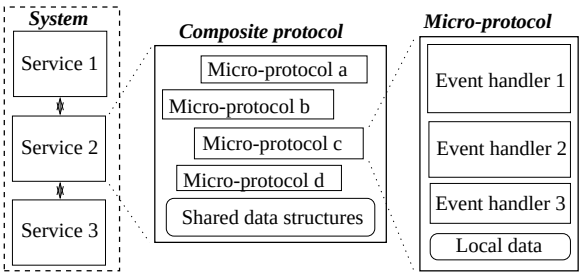

Figure 2: Two-level composition model

A micro-protocol is structured as a collection of *event handlers*, which are procedure-like segments of code *bound* to be executed when a specified *event* occurs. Events are used to signify state changes of interest, e.g., "message arrival from the network". When such an event occurs, all event handlers bound to that event are executed. Events can be *raised* by micro-protocols or be raised implicitly by the runtime system. Execution of event handlers is atomic with respect to concurrency, i.e., each handler is executed to completion without interruption. Further details on the model, event-handling operations, and other aspects of the approach can be found in [16, 17].

3.2 Design goals

The implementation of a configurable service has two major goals: minimizing execution overhead and maximizing degree of configurability. The execution overhead is the performance difference between a monolithic implementation of a service and a configurable implementation that provides equivalent guarantees. The design of the micro-protocols can impact the performance of a service considerably. Although the goal is to implement each property as an individual micro-protocol, for efficiency reasons, micro-protocols cannot be designed in isolation. For example, if the implementation of property p_i requires a site to multicast

a message and the implementation of property p_j has the same requirement, the implementations should only multicast one message, possibly with separate fields for p_i and p_j. Furthermore, redundant work in different micro-protocols should be avoided by having only one do the work, with the others utilizing those results. This can be accomplished, for example, by making the result available as global data and using events to notify other micro-protocols.

The degree of configurability is the number of different correct configurations of a service given a set of micro-protocols. The theoretical maximum degree of configurability is defined by all possible combinations of properties in the dependency graph. Note that, for practical reasons, an implementation of a configurable service often has a degree of configurability less than the theoretical maximum.

3.3 Relations between micro-protocols

Like their underlying properties, micro-protocols cannot be combined in arbitrary ways due to relations between micro-protocol. Let $imp(m, p)$ denote that micro-protocol m implements property p. As with properties, conflict and independence are two of the relations between micro-protocols, with definitions as follows:

- Micro-protocols m_1 and m_2 *conflict* if they cannot be configured into the same system, which may be the result of the corresponding properties conflicting or design decisions made during the implementation.
- Micro-protocols m_1 and m_2 are *independent* if m_1 can be used without m_2, m_2 can be used without m_1, and m_1 and m_2 can be used together, where the combination guarantees both the properties implemented by m_1 and m_2.

The third relation between properties, dependency, is divided when considering micro-protocols into separate relations called *dependency* and *inclusion*. To motivate this, consider two properties p_1 and p_2 such that $dep(p_1, p_2)$. Based on the definition of dependency between properties, any system that guarantees p_1 must also guarantee p_2. While sufficient for abstract properties, in an implementation it is useful to identify two possible ways in which this can be achieved. First, we could implement m_2 such that it realizes p_2, $imp(m_2, p_2)$, and then implement p_1 as micro-protocol m_1 that builds on the guarantees made by m_2. This means that for m_1 to operate correctly, m_2 must also be present, i.e., only the combination of m_1 and m_2 guarantees the service has property p_1. This approach preserves the dependency that exists between properties as a dependency between micro-protocols. The second alternative is to implement m_1 such that it implements both p_1 and p_2, while m_2 is implemented to only realize p_2. This approach creates micro-protocols m_1 and m_2 where m_1 is strictly stronger than m_2. In this case, we say that m_2 *includes* m_1.

The definitions and practical implications of dependency and inclusion relations for micro-protocols are as follows:

- Micro-protocol m_1 *depends on* m_2 if m_2 must be present in the configuration of a service and operate correctly in order for m_1 to provide its specified service. In practice, this means that if m_1 is to be configured into a service, m_2 must be configured in as well.

- Micro-protocol m_1 *includes* m_2 if m_1 implements a property strictly stronger than that implemented by m_2 without relying on m_2 being configured in the service. In practice, this means that m_1 and m_2 must not be used together.

The choice of implementation approach and the resulting relation between the micro-protocols is based primarily on convenience. Often, the implementation of property p_1 cannot take advantage of property p_2 being satisfied, even though the properties have a logical dependency relation. In this case, it is usually simpler to implement p_1 so that p_2 is satisfied directly, independent of the implementation of p_2. An example is the relation between orphan termination and interference avoidance. Although orphan termination formally depends on interference avoidance, its implementation cannot be built on the implementation of interference avoidance. As a result, the relation between the respective micro-protocols is an inclusion relation. In other situations, the implementation of a property can take advantage of the guarantees provided by some other micro-protocol. For example, the implementation of message ordering properties can exploit a reliability micro-protocol that ensures that all sites will eventually receive every message.

3.4 Configuration graphs

Configuration graphs are a graphical method of representing configuration constraints caused by relations between micro-protocols in much the same way that dependency graphs represent relations between properties. In this graph, nodes represent micro-protocols, directed edges dependencies, and unlabeled choice nodes conflicts. The inclusion relation is represented by node inclusion, i.e., if micro-protocol m_1 includes m_2, the node for m_2 in the configuration graph is inside the node for m_1. To simplify the graphs, we assume a micro-protocol inherits all the dependencies of the micro-protocols that it includes unless otherwise stated. The configuration graph should also identify the minimal set of micro-protocols required to implement the service. This can be done, for example, by having a virtual USER node with dependency edges to those micro-protocols that are required to implement a minimal service.

A configuration graph can be used as a tool for configuring customized services. The designer of a service first decides which properties are required and identifies the micro-protocols that implement those properties in the configuration graph. These micro-protocols are then included in the configuration, along with all micro-protocols on which the chosen ones depend. Any micro-protocol may be replaced by one that includes the original one. Only one micro-protocol from each choice node may be chosen and no micro-protocol m_1 such that micro-protocol m_2 is in the configuration and m_2 includes m_1 may be chosen.

A configuration graph of a highly-configurable group RPC service described in [16] is presented in figure 3. The goal of the group RPC implementation was to be highly configurable but not necessarily reach the theoretical maximum degree of configurability, and thus, the configuration graph has some additional configuration constraints compared to the corresponding dependency graph in figure 1. In particular, the ordering micro-protocols conflict with BoundedTermination even though the corresponding properties are independent. The reason is the way BoundedTermination is implemented in this micro-protocol suite. In particular, it terminates an RPC call at the given deadline by returning a failure indication to the client and marking the call as completed. This causes the ReliableCommunication micro-protocol to stop from

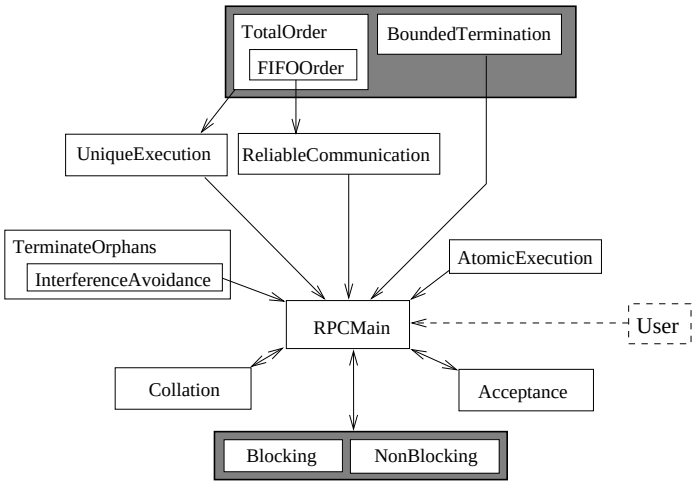

Figure 3: Group RPC Configuration Graph

retransmitting the call, which can cause a call to not reach all servers. The ordering micro-protocols rely on all messages eventually arriving at all servers and thus an omission of a message will cause a deadlock. Also, TotalOrder depends on UniqueExecution to filter out retransmissions of a call. It would have been possible to implement the micro-protocols without introducing these new constraints, but these simplifications made the design much easier.

Furthermore, an additional micro-protocol RPCMain is introduced in the design. This micro-protocol implements the basic behavior of an RPC service, i.e., it forwards requests and responses through the group RPC composite protocol. Other micro-protocols augment this basic behavior. Finally, note that all acceptance and collation properties are implemented by one acceptance and one collation micro-protocol. These micro-protocols are designed so that they can be parameterized to provide any desired acceptance or collation property. In particular, the acceptance micro-protocol can be passed a parameter that indicates how many responses must be received for a call to be considered successful and the collation micro-protocol can be passed any function that combines multiple responses.

Despite the increased number of dependencies and conflicts, this design still allows 132 semantically different correct configurations even without considering all the potential different collation and acceptance policies. Performance results of a number of the different configurations are presented in [15]. Given this set of micro-protocols, we can configure RPC service variants that realize the same set of properties as existing RPC services. For example, the basic RPC service described in [18] corresponds to a service configured with RPCMain, Blocking, Acceptance(1), Collation with a function that returns the first response, ReliableCommunication, and UniqueExecution. Rajdoot [19] corresponds to the same set, plus BoundedTermination and TerminateOrphans. Among group RPC services, the one-to-many RPC described in [20] corresponds to the set RPCMain, Blocking, Acceptance(N), Collation with a function consisting of identity and comparison to detect inconsistencies at the server processes, ReliableCommunication, UniqueExecution, and TotalOrder.

4 Configuration support tool

A configuration graph could be used manually to create correct configurations, but the configuration task can be simplified and mistakes reduced by using a configuration support tool that only allows correct configurations to be created. We are currently developing a tool that provides a graphical user interface for choosing micro-protocols and entering their parameters. The tool, illustrated in figure 4, presents the user with a list of available micro-protocols. A configuration is created by picking the desired micro-protocols from the list.

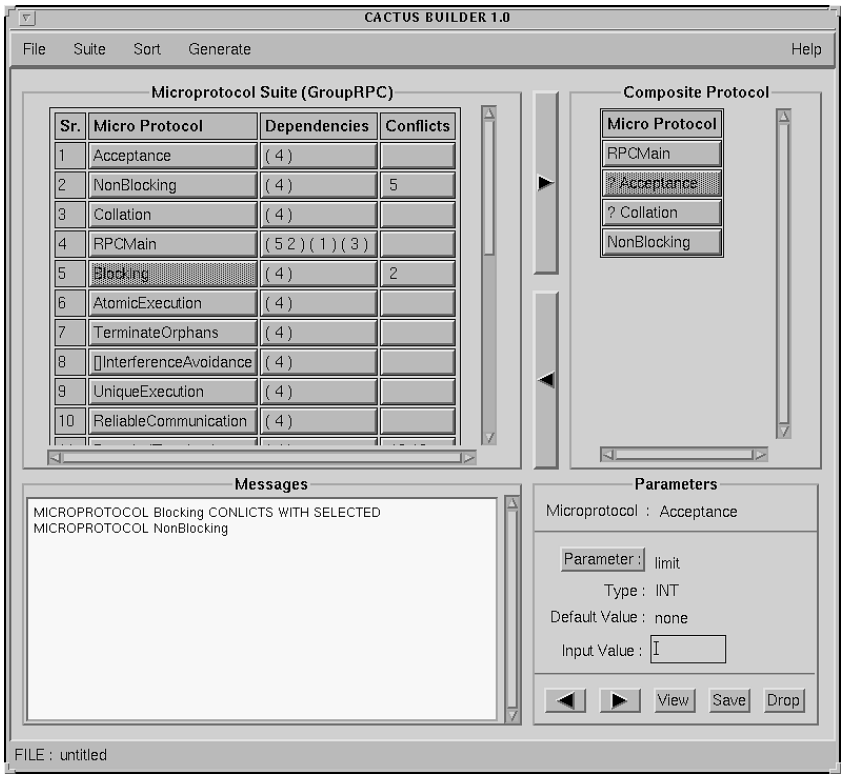

Figure 4: Configuration support tool user interface

The tool enforces the dependency, inclusion, and conflict relations between micro-protocols by textually instructing the user if another micro-protocol must be chosen to satisfy a dependency or if the chosen micro-protocol cannot be included because of a conflict or an inclusion. It also ensures that the user enters any required parameters for the selected micro-protocols. When all the necessary micro-protocols have been selected and the parameters entered, the tool can be used to generate all the files necessary to compile the chosen configuration. The tool allows a chosen configuration to be saved and later reloaded and modified. This makes it easy to make small changes to configurations.

The tool requires knowledge about the available micro-protocols, relations between them, and parameters needed by each micro-protocol. This information is provided in the form of a specification file that contains information about relations and parameters. For each parameter, the file includes name, type, default value,

a text description, and input type. The input type specifies if the parameter must be entered by the user or if the parameter is fixed. The specification file of the Acceptance micro-protocol is presented in figure 5. Note that for fixed parameters, the tool always uses the default value provided in the specification file.

```
MICRO_PROTOCOL {
        NAME : Acceptance ;
        VERSION : 1.0 ;
        DESC : Determines when a call can be accepted as successful, i.e.,
                when the required number of responses has been received;
        DEPENDS_ON : RPCMain ;
        CONFLICTS_WITH : NONE ;
        INCLUDES : NONE ;
        PARAMETER {
                NAME : grpc ; TYPE : FIXED POINTER ; DEFAULT : (this) ;
                DESC : A pointer to the GRPC composite protocol ;
        }
        PARAMETER {
                NAME : limit ; TYPE : INPUT INT ; DEFAULT : 1 ;
                DESC : Number of responses required before a call is considered successful ;
        }
}
```

Figure 5: Specification file of the Acceptance micro-protocol

5 Configuration management in other systems

Although our configuration management methodology has been developed in the context of the micro-protocol approach, the fundamental relations, as well as the configuration graphs, can be applied to any modular configurable systems. To illustrate this, we take a look at four other popular approaches to constructing configurable software and how the relations between software components apply in these cases.

5.1 Hierarchical approaches

A system is constructed as a stack, or directed graph, of modules. Each module typically interacts only with modules immediately above and below it in the hierarchy. Recent examples of this approach are the *x*-kernel [11], Horus [14], the Genesis database system [8], and stackable file systems [5]. Some of the approaches require that all modules export an identical interface, e.g., *x*-kernel, while others allow layer-specific interfaces, e.g., Genesis.

In the systems where all modules export the same interface, any stack of modules would be syntactically valid. However, in practice, semantic constraints between modules dictate that only certain combinations are correct. In particular, some modules require certain other modules underneath them, i.e., depend on the

other modules [21]. Furthermore, often there is a choice of protocols, or modules in general, for a certain layer of the hierarchy. This translates naturally to conflict since only one of these protocols can typically be chosen in any configuration. Thus, both dependency and conflict relations typically appear in such systems. Without a tool such as the configuration graph, the designer of the modules has to convey these restrictions to service users in an ad hoc manner. A configuration graph can be used to represent such constraints as illustrated in figure 6. In this figure, there are three logical layers of modules, with a choice of modules in each layer. Note, however, that only the lowest layer must be chosen, i.e., the two higher layers are optional. Furthermore, the second layer may be omitted in some cases depending on which module is chosen in layer 3.

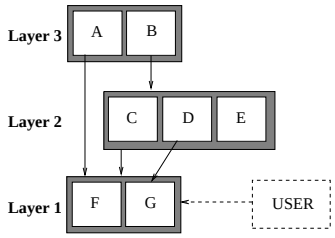

Figure 6: Configuration graph of a hierarchical system

Although we are able to use a configuration graph to describe configuration constraints between software modules in hierarchical systems, our configuration support tool could not be used with x-kernel, Horus, or the other systems without some modifications. In particular, all the files that the tool generates are very system, platform, and programming language specific. Furthermore, some additional information, such as the relative order of the modules in the hierarchy, may have to be added to the specification files for the tool to generate the proper files.

5.2 Slotted approaches

A customizable software component is constructed as a fixed system backplane with slots that can be filled using a choice of modules for each slot. Examples of this approach are Adaptive [12] and a framework for group communication systems presented in [22].

In contrast with the hierarchical approach, modules in this approach are typically typed and thus can only be used in one specific slot. However, there may be other relations between modules that typing of modules does not capture. The configuration graph approach can be easily applied for slotted systems as illustrated in figure 7. Each slot in the system must be filled with a chosen module, thus the backplane module and each of the sets of modules depend on one another. However, there may be some extra dependencies, such as if module C is chosen, module A must be chosen as well. In some systems, e.g., Adaptive, a slot can be filled with a *composite* module. The composite module is essentially a smaller backplane with new slots. In the figure, module F is a composite module. Note that the slotted approach may appear to be similar to the hierarchical approach if we just rotate the figure 90 degrees. There are some major differences, however,

since the hierarchical approach allows a choice of number of layers and often makes it easy to add new layers between existing ones, whereas it is usually more difficult to add new slots.

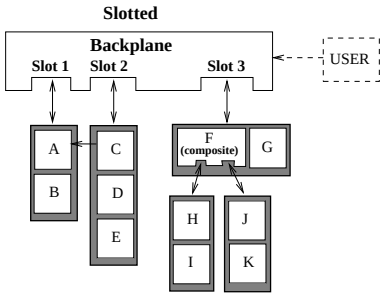

Figure 7: Configuration graph of a slotted system

5.3 Class hierarchy based approaches

In this approach, the mechanisms for constructing a customized software component are presented to users as an object class hierarchy. A predefined class hierarchy specifies the available components, which can be manipulated by invoking the object methods. New classes can be defined as derived classes of existing ones. Examples of this approach are Arjuna [23] and a configurable mixed-media file system [24].

The subclass relation between modules is well-defined in the class hierarchy based approach. This relation maps naturally to inclusion: a subclass augments its base class and by definition includes the base class. However, other relations such as dependencies are typically not captured by the subclass relation and therefore the configuration graphs could be a valuable tool also for systems designed using this approach (see figure 8).

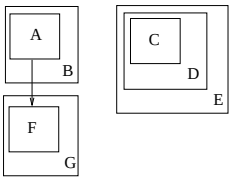

Figure 8: Configuration graph of a class hierarchy based system

5.4 Object-message approaches

A system consists of a set of objects that communicate with each other by message passing. Examples of this approach are the Actor model [25], where a computation is constructed of actor objects that communicate using point-to-point messages, and the model for parallel execution of communication protocols presented in [26].

All the four relations between modules may be present in systems designed using this approach. If object A sends a message to object B, there may be a dependency between them. Note that if messages are the only allowed form of interaction between objects, there can only be direct dependencies between objects that communicate with one another. Note also that if both the sender and receiver need to name one another to exchange a message, they both depend on one another. However, some of these approaches allow more flexible forms of communication such as multicasts, which do not imply mutual dependency. In addition to dependencies, there may be alternative implementations of certain objects and these objects may have the conflict or inclusion relations. An example of conflict would be different algorithms for implementing total order and an example of inclusion would be a causal ordering object that includes a fifo ordering object. Finally, objects may also be independent, for example, an encryption object could be independent of an ordering object. Due to the flexibility of this approach, it is hard to provide a representative example of a configuration graph for this approach. However, figure 9 provides one example with a number of the different relations present.

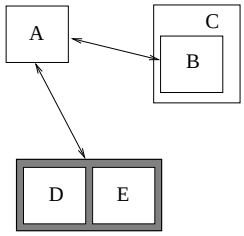

Figure 9: Configuration graph of an object-message based system

6 Related work

The concept of configuration management has been applied in a number of different contexts ranging from managing the internal configuration of one software component to managing the configuration of a computer system consisting of hardware and software components. Since our emphasis is naturally on software, we focus on the related work in this area.

The most widely applied form of configuration management is *software configuration management* (SCM), which is often also called version, source, or revision control. SCM involves managing the development of the components from which a complex system is built and integrating these components to form that system. SCM has been extensively studied [27], standards have been published (IEEE Std 828-1990, NASA D-GL-11, DoD MIL-STD-973), and a number of commercial tools have been developed. The main thrust of SCM is managing the different versions of the software modules in a complex system. Forms of dependency and conflict between software modules are supported in SCM. However, dependencies between software modules in SCM typically reflect the implementation, i.e., if software module A imports (or "includes") software module B then A depends on B, rather than semantic dependencies. Similarly, conflict or choice is primarily between different versions of a module—e.g., version 1.0, 1.1—or between different

implementations of the module for different hardware platforms—e.g., Linux version, Windows NT version.

The main difference between our work and SCM is that we support configuration management for software out of which hundreds, potentially thousands, of different customized variants may be created, whereas the goal of SCM is to support the creation of a few instances of a complex systems and the evolution of such a system. There has been some work on extending SCM to provide customized features for different customers. For example, the Harmony project [28] supports configuration-specific inclusion files that list the source files to be included in a given configuration and thus allows different configurations to be created. However, even this tool does not identify or support semantic relations between modules.

Another area of configuration management is *configuration programming*, defined as the process whereby components written in any conventional programming language can be bound together to form a dynamic system [29]. Examples of this work are Darwin [30], Polylith [31], and UNICON [32]. In contrast to our work, the primary goal of these systems is to make it possible to combine existing software components into a larger system primarily by linking the inputs and outputs of the components using pipes, RPC, or shared data structures. The main challenge is to get the I/O of the components to match syntactically. In our approach, we are assuming the modules of a customizable software component are designed to be combined with one another so no extra work is required to bind them together. However, if this was not the case, the configuration programming languages could be used in conjunction with our work to make is possible to bind independently developed modules together. Furthermore, our approach could be used in conjunction with the configuration programming languages to capture the semantic relations between the components.

Finally, there is work on configuration management aiming at determining if a configuration is consistent based on syntactic and semantic compatibility of interacting modules. For example, [33] uses predicates to describe the acceptable inputs to a module (preconditions) as well as its outputs (postconditions). A configuration is consistent if for every communicating pair of modules, the postconditions of the output satisfy the preconditions of the input. The approach has been extended to handle dynamic system configuration changes as a result of new modules being introduced [34]. This is somewhat orthogonal to our work since we assume the modules of a configurable software component are designed and implemented so that they operate correctly together (independence or dependency) or they must not be used together (conflict or inclusion). However, techniques such as these could be useful in detecting errors in the design of the modules and in the configuration graph. Unfortunately, these techniques do not apply directly to the micro-protocol approach, since the interactions between micro-protocols are through events and shared data structures and thus much more flexible, as well as much more difficult to analyze, than strict communication interactions.

7 Conclusions and future work

This paper has presented a methodology that simplifies the complex task of constructing customized instances of highly-customizable software. The methodology is based on understanding the relations between software modules that dictate which combinations will execute correctly. We are developing a configuration support tool that enforces these relations and thus allows only correct configurations to be created. Although the

methodology has been developed for our micro-protocol approach, the relations as well as the tools could be applied for other types of modular configurable systems.

Our future work on configuration management has two major aspects. First, in addition to the relations that determine which combinations of modules are correct, modules have more subtle relations where one module may affect the performance or real-time guarantees made by another module. Our goal is to understand such relations and the underlying tradeoffs between service properties. Second, we intend to develop a higher-level configuration support tool that allows the service user to state his requirements in terms of abstract service properties rather than needing to choose from a set of software modules.

Acknowledgments

This work is based on the author's dissertation work done under the supervision of Prof. Richard Schlichting whose guidance has been invaluable. Sameer Verkhedkar designed and implemented the configuration support tool. An earlier version of this paper appeared in [35].

References

[1] C. Pu, H. Massalin, and J. Ioannidis. The Synthesis kernel. *Computing Systems*, 1(1):11–32, 1988.

[2] B. Bershad, S. Savage, P. Pardyak, E. Sirer, M. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility, safety, and performance in the SPIN operating system. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pages 267–284, Copper Mountain Resort, CO, Dec 1995.

[3] D. Engler, M. Kaashoek, and J. O'Toole. Exokernel: An operating system architecture for application-level resource management. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pages 251–266, Copper Mountain Resort, CO, Dec 1995.

[4] A. Veitch and N Hutchinson. Dynamic service reconfiguration and migration in the Kea kernel. In *Proceedings of the 4rd International Conference on Configurable Distributed Systems*, pages 156–163, Annapolis, MD, May 1998.

[5] J. Heidemann and G. Popek. Performance of cache coherence in stackable filing. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pages 127–142, Copper Mountain Resort, CO, Dec 1995.

[6] Y. Khalidi and M. Nelson. Extensible file systems in Spring. In *Proceedings of the 14th Symposium on Operating Systems Principles*, Asheville, NC, Dec 1993.

[7] O. Krieger and M. Stumm. HFS: A performance-oriented flexible file system based on building block composition. *ACM Transactions on Computer System*, 15(3):286–321, Aug 1997.

[8] D. Batory, J. Barnett, J. Garza, K. Smith, K. Tsukuda, B. Twichell, and T. Wise. GENESIS: An extensible database management system. *IEEE Transactions on Software Engineering*, SE-14(11):1711–1729, Nov 1988.

[9] M. Stonebraker and L. Rowe. The design of Postgres. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 340–355, 1986.

[10] P. Schwarz, W. Chang, J. Freytag, G. Lohman, J. McPherson, C. Mohan, and H. Pirahesh. Extensibility in the Starburst database system. In *Proceedings of the International Workshop on Object-Oriented Database Systems*, pages 85–93, Asilomar, CA, Sep 1986.

[11] N. Hutchinson and L. Peterson. The *x*-kernel: An architecture for implementing network protocols. *IEEE Transactions on Software Engineering*, 17(1):64–76, Jan 1991.

[12] D. Schmidt, D. Box, and T. Suda. ADAPTIVE: A dynamically assembled protocol transformation, integration, and evaluation environment. *Concurrency: Practice and Experience*, 5(4):269–286, Jun 1993.

[13] S. Mishra, L. Peterson, and R. Schlichting. Consul: A communication substrate for fault-tolerant distributed programs. *Distributed System Engineering*, 1:87–103, Dec 1993.

[14] R. van Renesse, K. Birman, and S Maffeis. Horus, a flexible group communication system. *Communications of the ACM*, 39(4):76–83, Apr 1996.

[15] N. Bhatti, M. Hiltunen, R. Schlichting, and W. Chiu. Coyote: A system for constructing fine-grain configurable communication services. *ACM Transaction on Computer Systems*, Nov 1998.

[16] M. Hiltunen. *Configurable Distributed Fault-Tolerant Services*. PhD thesis, Dept of Computer Science, University of Arizona, Tucson, AZ, Jul 1996.

[17] N. Bhatti. *A System for Constructing Configurable High-Level Protocols*. PhD thesis, Dept of Computer Science, University of Arizona, Tucson, AZ, Nov 1996.

[18] A. Birrell and B. Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1):39–59, Feb 1984.

[19] F. Panzieri and S. Shrivastava. Rajdoot: A remote procedure call mechanism supporting orphan detection and killing. *IEEE Transactions on Software Engineering*, SE-14(1):30–37, Jan 1988.

[20] E. Cooper. Replicated distributed programs. In *Proceedings of the 10th ACM Symposium on Operating Systems Principles*, pages 63–78, Orcas Island, WA, 1985.

[21] R. van Renesse, K. Birman, R. Friedman, M. Hayden, and D. Karr. A framework for protocol composition in Horus. In *Proceedings of the 14th ACM Principles of Distributed Computing Conference*, pages 80–89, Aug 1995.

[22] R. Golding. *Weak-Consistency Group Communication and Membership*. PhD thesis, Dept of Computer Science, University of California, Santa Cruz, Santa Cruz, CA, Dec 1992.

[23] S. Shrivastava, G. Dixon, and G. Parrington. An overview of the Arjuna distributed programming system. *IEEE Software*, 8(1):66–73, Jan 1991.

[24] S. Maffeis. Design and implementation of a configurable mixed-media file system. *Operating Systems Review*, 28(4):4–10, Oct 1994.

[25] G. Agha. Concurrent object-oriented programming. *Communications of the ACM*, 33(9):125–141, Sep 1990.

[26] K. Schwan, T. Bihari, and B. Blake. Adaptive, reliable software for distributed and parallel real-time systems. In *Proceedings of the 6th IEEE Symposium on Reliability in Distributed Software and Database Systems*, pages 32–42, Mar 1987.

[27] E. Bersoff, V. Henderson, and S. Siegel. *Software Configuration Management*. Prentice-Hall, 1980.

[28] W. Gentleman, S. MacKay, D. Stewart, and M. Wein. Commercial realtime software needs different configuration management. In *Proceedings of the Second International Workshop on Software Configuration Management*, pages 152–161, Princeton, NJ, Oct 1989.

[29] J. Bishop and R. Faria. Connectors in configuration programming languages: Are they necessary. In *Proceedings of the 3rd International Conference on Configurable Distributed Systems*, pages 11–18, Annapolis, MD, May 1996.

[30] J. Magee, N. Dulay, and J Kramer. Regis: A constructive development environment for distributed programs. *Distributed Systems Engineering Journal*, 1(5):304–312, Sep 1994.

[31] J. Purtilo. The Polylith software bus. *ACM Transactions on Programming Languages and Systems*, 16(4):151–174, Jan 1994.

[32] M. Shaw, R. DeLine, D. Klein, T. Ross, D. Young, and G. Zelesnik. Abstractions for software architecture and tools to support them. *IEEE Transactions on Software Engineering*, 21(4):314–335, Apr 1995.

[33] D. Perry. Software interconnection models. In *Proceedings of the 9th International Conference on Software Engineering*, pages 61–69, 1987.

[34] P. Feiler and J. Li. Consistency in dynamic reconfiguration. In *Proceedings of the 4th International Conference on Configurable Distributed Systems*, pages 189–196, Annapolis, MD, May 1998.

[35] M. Hiltunen. Configuration management for highly-customizable services. In *Proceedings of the 4th International Conference on Configurable Distributed Systems*, pages 197–205, Annapolis, MD, May 1998.