Eliminating Execution Overhead of Disabled Optional Features in Connectors

Lubomír Bulej[1,2] and Tomáš Bureš[1,2]

[1] Distributed Systems Research Group, Department of Software Engineering
Faculty of Mathematics and Physics, Charles University
Malostranské nám. 25, 118 00 Prague, Czech Republic
phone +420-221914267, fax +420-221914323
{bulej,bures}@nenya.ms.mff.cuni.cz

[2] Institute of Computer Science, Academy of Sciences of the Czech Republic
Pod Vodárenskou věží 2, 182 07 Prague, Czech Republic
phone +420-266053831

Abstract. Connectors are used to realize component interactions in component systems. Apart from their primary function, which is mediating the communication, their implementation can also support additional features that, while unrelated to the primary function, may benefit from their placement in connectors. Such features are often optional in the sense that they can be activated and deactivated at run-time. The problem is that even if they are disabled, their very presence in the connector incurs certain overhead. In this paper, we describe an approach to eliminate this overhead by reconfiguration of the connector implementation. Besides connectors, the approach is applicable to similar technologies such as reflective middleware and other architecture-based component systems and frameworks.

Keywords. Component systems, software connectors, runtime reconfiguration.

1 Introduction

In component systems with support for distribution, the design-time connections among components usually represent a more powerful concept than just a plain reference as known from programming languages. Such connection, a hyper-edge in general, connects components that participate in some kind of interaction. The endpoints of a connection correspond to the roles the connected components assume in the interaction.

At runtime, besides entities implementing the components, additional entities are required to realize the interaction modeled by such connections. The exact composition of entities required to implement a given interaction depend on the communication style governing the interaction. In case of e.g. procedure call, these entities could be a stub and a skeleton, or a plain reference, depending on the location of the participating components.

To model the component interactions, many component systems have introduced connectors as first-class design entities that model the interactions among components. At the design level, connectors represent a high-level specification of the interactions they model, aggregating requirements imposed on the interaction. The specification is then used for transformation of a design-time connector into runtime connector, which implements the interaction and has to satisfy the requirements from the specification.

Some of the requirements may not be directly related to the primary function of a connector, which is to enable communication among components. The properties a runtime connector needs to have to satisfy such requirements are called *non-functional properties*. A connector implementation gains such properties by implementing additional functionality unrelated to its primary function, such as logging, performance and behavior monitoring, security[3], etc.

Some of the features implemented by a connector to satisfy connection requirements may be *optional*, which means that they do not need to be active at all times. The implementation of a connector may support selective activation and deactivation at runtime, especially if a feature incurs considerable *dynamic overhead* when active.

The problem is that even when an optional feature is disabled, its presence in a connector may incur certain *static overhead*. The overhead is caused by the mechanism used to integrate an optional feature with other code. Compared to its dynamic overhead, static overhead of an optional feature tends to be small or even negligible, depending on the integration mechanism. Nevertheless, the existence of the static overhead and the lack of data quantifying its impact is often a reason for not including useful features such as logging or monitoring in production-level applications. Such features can provide an application administrator with tools for diagnosing problems in a running application.

As discussed in [1], due to complexity of contemporary software, problems and misbehavior occurring in production environment, which typically does not provide sufficient diagnostic tools, are hard to reproduce in development environment where the tools are available. For this reason, even production-level applications should always provide features that can be activated at runtime and that can assist in diagnosing hardly reproducible problems. Moreover, as long as they are not used, those features should have no impact on the execution of the application.

In this paper, we present an approach to eliminate static execution overhead of disabled optional features in the context of one particular model [4] of architecture-based connectors. The model is being developed within our research group with focus on automatic generation of connector implementation from the high-level specification. The issues concerning efficiency of generated connectors have prompted the research presented in this paper.

[3] Security properties such as authentication or encryption cannot be considered entirely unrelated, but are still considered non-functional, because they are not essential.

Even though we present it on a specific connector model, the scope in which the approach can be applied is much broader. The principal requirements are construction through composition and the ability to track dependencies in a composite at runtime. These requirements are satisfied even in very simple component environments, therefore the presented approach is directly applicable e.g. to component-based middleware (e.g. the reflective middleware by Blair et al. [2], Jonathan [3], etc.), which in fact plays the role of connectors.

1.1 Goals of the Paper

The main goal is to eliminate the static execution overhead associated with optional connector features that have been disabled. This will allow including optional features such as logging, or monitoring even in production-level applications without impacting performance while the features are disabled.

To solve the problem in the context of architecture-based connectors, we need an algorithm that allows us to propagate changes in the runtime structure of a connector implementation through the connector architecture. This in turn cannot be done without imposing certain requirements on the connector runtime.

The goals are therefore to devise an algorithm for propagating changes in a connector architecture, to formulate the requirements that a connector runtime must satisfy for the algorithm to work, and to prove that the algorithm always terminates in a finite number of steps.

1.2 Structure of the Text

The rest of the paper is organized as follows: Section 2 gives an overview of the connector model used as a testbed for our approach, Section 3 provides discussion of the overhead associated with optional features with respect to the connector model, Section 4 outlines the solution and Section 5 describes in detail the proposed changes in connector runtime and the reconfiguration process used to eliminate the static execution overhead of disabled optional features, Section 6 provides an overview of related work, and Section 7 concludes the paper.

2 Connector Model

Throughout the paper, we use a connector model described in our earlier work [4] as a test bed. The model has a number distinguishing features, which are not commonly present in other connector models. Connectors are modeled by connector architectures describing composition of entities with limited, but well-defined functionality. The key feature with respect to the approach presented in this paper is that the design architecture of a connector is reflected in runtime architecture of the connector implementation and can be traversed and manipulated at runtime.

2.1 Connectors from the Outside

Conceptually, a connector is an entity exposing a number of attachment points for components participating in an interaction. Technically, due to its inherently distributed nature, a connector comprises a number of *connector units*. Each unit represents a part of a connector that can exist independently in a *deployment dock*, which hosts instances of components and connectors. A connector unit communicates (strictly) locally with components attached to it and remotely (using middleware) with other connector units.

An example in Figure 1 shows client components A, B, and C connected to the Server component using connectors. In case of components B and C the respective connectors cross the distribution boundary between address spaces.

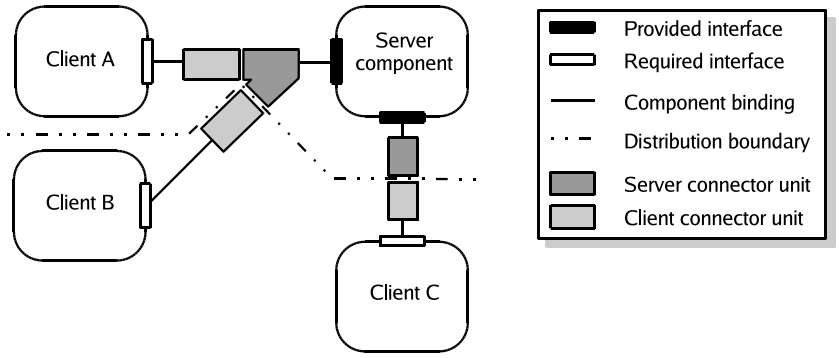

Fig. 1. Using connectors to mediate communication among components

2.2 Connectors from the Inside

The construction of connectors is based on hierarchical composition of well-defined entities with limited functionality. A connector is made of connector units, and connector units are made of *connector elements*. The composition of connector elements is described by *connector architecture*.

Since the connector elements can be nested, the entire architecture forms a hierarchy with the connector as a whole represented by its root. The internal nodes represent composite elements which can contain other elements, and the leaves represent primitive elements which encapsulate implementation code. Connector units at the second level of the hierarchy are also connector elements, except with certain restrictions on bindings among other elements.

While the connector architecture models composition of *connector element types* [4], *connector configuration* provides a white-box view of a connector, which is obtained from the architecture by assigning concrete implementation to the

element types present in the architecture. A connector configuration therefore fully determines the connector implementation and its properties.

An example of a connector configuration is given in Figure 2, which shows a connector realizing a remote procedure call. The connector consists of one server unit and one client unit. The client unit consists of an *adaptor* (an element realizing simple adaptations in order to overcome minor incompatibility problems) and a *stub*. The server unit comprises a *logger* (element responsible for call tracing), a *synchronizer* (element realizing a specific threading model), and a *skeleton collection* (element which groups together multiple skeleton implementation using different middleware, thus enabling access to the server unit via different protocols).

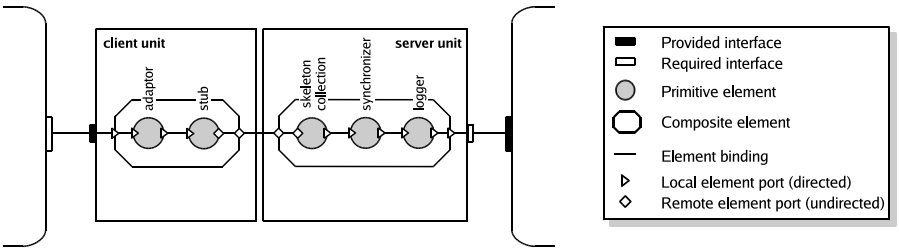

Fig. 2. Connector at runtime

A connector element communicates with other elements or with a component interface only through designated ports. There are three basic types of element ports in the model: (a) *provided ports*, (b) *required ports*, and (c) *remote ports*. Based on the types of ports connected together we distinguish between local (required-to-provided or provided-to-required ports) and remote (between multiple remote ports) bindings.

Local bindings are realized via local calls, which limits their use to a single address space. Ports intended for local bindings thus serve for (a) element-to-element communication within a connector unit and (b) element-to-component communication with the component attached to a respective connector unit.

Remote bindings represent a complex communication typically realized by middleware (e.g., RMI, JMS, RTP, etc.). We do not attempt to model this communication or capture the direction of the data flow in our connector model – instead we view these bindings as undirected. To support other communication schemes than just point-to-point (e.g., broadcast), we model a remote binding as a hyper-edge which connects multiple remote ports.

The exact implementation of a remote binding depends on the participating elements. Their responsibility is to provide remote references or use remote references for establishing a connection. From the point of view of the connector model, a remote binding only groups together ports of elements sharing the same set of remote references.

Due to inherently distributed nature of a connector, there are restrictions on the occurrence of local and remote ports in the architecture and the bindings among them. At the top level of the architecture, a connector can only expose local ports. Remote bindings can only occur at the second level of the architecture, i.e. among connector elements representing connector units. In composite elements, only local bindings between child elements, and delegations between the child and parent element ports are allowed. Remote port occurring in a non-unit element must be delegated to the parent element.

2.3 Connectors at Runtime

At runtime, each connector element is represented by a primary class which allows the element to be recognized and manipulated as an architectural element. Depending on the types of declared ports, the primary class has to implement the following interfaces: *ElementLocalClient* (if the element has a required port), *ElementLocalServer* (if the element has a provided port), *ElementRemoteClient* (if the element has a remote port and acts as a client in a remote connection), and *ElementRemoteServer* (if the element has a remote port and acts as a server in a remote connection). The primary class aggregates the control interfaces that can be used for querying references to element ports (server interfaces) and for binding ports to target references (client interfaces). The signatures of the control interfaces are shown in Figure 3.

2.4 Optional Features in Connectors

Since the local bindings among connector elements are realized by local calls, a sequence of connector elements with complementary ports with the same interface may result in a chain of elements through which method invocations have to pass. From this point of view, optional features utilizing interception or filtering to perform their function will fit well in the connector model. Interception is used to include activity related to invocation of specific methods in the call path, while filtering operates on the content passed along the call path.

Examples of optional features implemented through interception are logging, performance or behavior monitoring, or any other function requiring method-level interception, such as the stub and skeleton. Features implemented through filtering may include encryption, compression, and other data transformations. Figure 2 shows a connector architecture with an optional logging feature implemented by the *logger* element.

3 Overhead of Optional Features

The activity of optional features is the source of dynamic memory and execution overhead. The overhead tends to be significant, but is only present when an optional feature is enabled. For this reason, the dynamic overhead is not further analyzed in this paper.

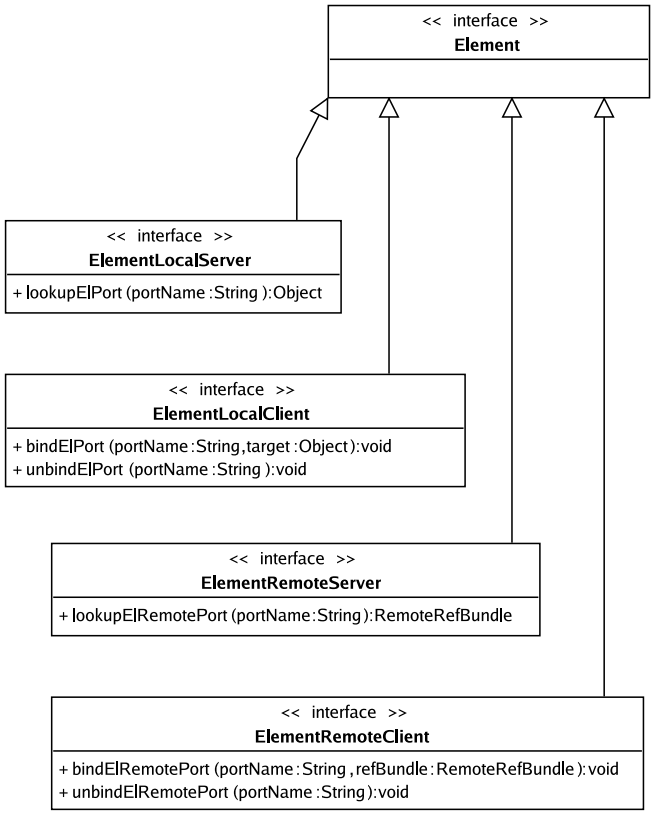

Fig. 3. Interfaces implemented by an element

The presence of an optional feature in a connector (or an application in general) is the source of static memory and execution overhead. The overhead is caused by the mechanism used to optionally include the implementation of a feature in a connector and by itself it tends to be rather small, even insignificant. The approach presented in this paper aims to address situations where these isolated cases accumulate.

3.1 Static Memory Overhead

The static memory overhead associated with optional features in connectors is caused by the additional code required to implement the desired operation. Eliminating the static memory overhead of a disabled optional feature requires eviction of the code that implements it from memory.

The ability to evict code from memory depends on the environment and may not be always possible. In case of virtual machine environment with garbage

collection such as Java, the code cannot be evicted explicitly – we can only attempt to ensure that it can be garbage-collected.

Considering the relatively light-weight connector runtime and the nature of the optional features, we assume the code implementing them will represent only a small fraction of all application code. Therefore we expect the static memory overhead to be negligible and do not specifically target it in our approach.

3.2 Static Execution Overhead

The static execution overhead associated with optional features is tied to the mechanism through which these features are included in the application call paths. This mechanism is based on invocation indirection, which is inherent to the connector model we are using (and to all component environments where the architecture is preserved at runtime). The concept of a connector architecture consisting of connector elements allows us to compose connectors from smaller blocks providing a clearly defined function, while the hierarchy serves to limit an element's awareness of its surroundings.

This allows adding optional features to connectors, because each element implementing a pair of complementary ports of the same type can serve as a proxy for another element. This is the case of the *logger* element in Figure 2. However, if multiple elements implementing optional features are chained together, the method invocation will have to go through all the elements in the chain, even if all optional features in these elements are disabled.

One may argue that in case of a single element, the execution overhead of one additional layer of indirection is negligible and can be tolerated. However, as mentioned above, connector elements are simple building blocks intended for composition. Therefore we expect connector architectures combining multiple optional features to achieve that through combination of multiple connector elements. In such case, the method invocation will accumulate overhead in each element of the chain, because before passing the invocation to the next element, each element must also decide whether to invoke the code implementing the feature. The overhead thus accumulated most probably will not be prohibitive, but it may not be negligible anymore.

4 Outline of the Solution

Eliminating the static execution overhead associated with a disabled optional feature on a specific application component interface requires removal of an element implementing the feature from a chain of elements intercepting method invocations on that interface.

This operation is similar to removal of an item from a single-linked list – the predecessor of the item must be provided with a link to its new successor. However, in case of connector architecture, the list items are not data but code. The problem thus gets complicated by the fact that connectors are distributed entities, and that each connector element is autonomous in deciding (anytime during

execution) when it wants to be removed from the call chain and when it wants to be part of it again. Additionally, the connector architecture is hierarchical, which further complicates the management of the call chain.

Connector elements in a chain intercepting particular component interface are linked to each other using a pair of complementary ports. Their required ports are bound to ports of the same type provided by their successors. Since each element is aware of its internal architecture but not of its place in the surrounding architecture, the binding between ports of two sibling elements must be established by the parent element. Based on its architecture, the parent element queries the child elements for references to their provided ports and provides these references to other child elements that require them.

Thus, when an element wants to be excluded from a call chain, it can simply realize it by providing the target reference associated with its required port as a reference to its own provided port. In other words, when queried for a reference to its provided port, an element returns a reference it has already obtained as a target for one of its required ports. Method invocations on the provided port are thus passed directly to the next element in the call chain.

Although the trick for excluding elements from the call path is simple, there are several problems that complicate its usage (a) for initial setup of connector architecture at startup and (b) for reconfiguration at runtime, because of the above mentioned connector element autonomy.

When creating the initial architecture (with some of the optional features disabled by default), it is necessary to bind the elements in certain order for the idea to work. The order would correspond to a breadth-first traversal of a graph of dependencies among the ports. The binding has to be done recursively for all levels of the connector architecture hierarchy, because through delegation and subsumption between the parent and child elements, the (virtual) graph of dependencies between the ports may cross multiple levels of the connector architecture hierarchy. Additionally, since the concept of connector elements is based on strong encapsulation, the child elements appear to their parent entity as a black-box. Consequently, there is neither central information about the connector architecture as a whole (viewed as a white-box), nor is it possible to explicitly construct the dependency graph for the entire connector.

In our approach, we address both mentioned situations by a reconfiguration process initiated within an element. The reconfiguration process starts when an event occurs that causes a reference to a provided port (exposed by an element) to be no longer valid and it is necessary to instruct all neighboring elements using this reference (through their required ports) to obtain an updated one. Because an element where such even occurs does not have information about its neighbors, it notifies its parent entity (the containing element or the connector runtime), which uses its architecture information to find the neighboring elements that communicate with the originating element, and are thus affected by the change of the provided reference. If there is a delegation leading to the originating element, the parent entity must also notify its own parent entity.

This reconfiguration process may trigger other reconfigurations when a depending element is excluded from the call chain – a change of the target reference on its required port causes a change of a reference to its provided port.

The reconfiguration process addresses the two situations – (a) initial setup of the connector architecture and (b) runtime reconfiguration – in the following way. In case of (a) we do not impose any explicit ordering on instantiation and binding of connector elements. When an element that wants to be excluded from the call chain is asked to provide a reference which it does not have yet (because its required port has not been bound yet), it returns a special reference *UnknownTargetReference*. As soon as the required port gets bound and the target reference (that should have been returned for the provided port) becomes known, the element initiates the reconfiguration process for the affected provided port, which ensures propagation of the reference to the affected elements.

In (b) the inclusion/exclusion of an element in/from a call chain affects a reference provided by a particular provided port – either a reference to an internal object implementing the port (including the optional feature code in the call path) or a target reference of a particular required port should be provided. In both cases the change is propagated to the depending neighbor elements through the reconfiguration process.

5 Reconfiguration Process

The reconfiguration process outlined in the previous section allows us to eliminate the static execution overhead of disabled optional features in connectors. In this section we show what extensions must be introduced to the connector runtime and what functionality must be added to the element control interfaces presented in Section 2.3.

Additionally, we show that the algorithm always terminates, which is not an obvious fact due to reconfiguration process triggering other reconfigurations. Due to space constraints, we have omitted additional discussion of the algorithm. The discussion, along with a more detailed description of the algorithm and the proof of termination can be found in [5].

5.1 Reconfiguration Algorithm

The reconfiguration algorithm is executed in a distributed fashion by all entities of the connector architecture. Because it operates on a hierarchical structure with strong encapsulation, each participant only has local knowledge and a link to its parent entity to work with.

The link is realized through *ReconfigurationHandler* interface, which is implemented by all non-leaf entities of the connector architecture, i.e. composite elements and connector runtime, and is provided during instantiation to all non-root entities, i.e. composite and primitive elements. When the reconfiguration process needs to traverse the connector architecture upwards (along the delegated ports), the respective connector element uses that interface to initiate

the reconfiguration process in its parent entity. In the case of our connector model, the *ReconfigurationHandler* interface contains two methods that serve for invalidating provided and remote server ports of the element initiating the reconfiguration process.

When disabling an optional feature, the element providing the feature remains in the connector architecture but is excluded from a call path passing through its ports. As described in Section 4, when an element wants to be excluded from the call chain, it provides target reference associated with its required port as a reference to its provided port. This constitutes an internal dependency between the ports which the reconfiguration process needs to be able to traverse. Since this dependency is not allowed between all port types, we enumerate the allowed cases and introduce the notion of a *pass-through port*.

An element port is considered pass-through, iff one of the following holds:

1. the port is provided and is associated with exactly one required port of the same element; if the reference to the port is looked up, the target of the associated required port is returned instead,
2. the port is provided and is associated with exactly one remote client port of the same element; if the reference to the port is looked up, a single target from the reference bundle of the associated remote port is returned instead,
3. the port is remote server and is associated with one or more required ports of the same element, and if the reference to the port is looked up, the returned reference bundle contains also the targets of the associated required ports.

Pass-through target is a target object of a required port that is associated with a pass-through port and to which the invocations on the pass-through port are delegated.

The reconfiguration process can be initiated at any non-root entity of the connector architecture, whenever a connector element needs to change a reference to an object implementing its provided or remote server port. This can happen either when an element's required or remote client port associated with a pass-through port is bound to a new target, or when an element needs to change the reference in response to an external request.

The implementation of each element must be aware of the potential internal dependencies between the pass-through ports and their pass-through targets. Whenever a change occurs in an element that changes the internal dependencies, the element must initiate the reconfiguration process by notifying its parent entity about its provided and remote server ports that are no longer valid.

The implementation of the reconfiguration algorithm is spread among the entities of the connector architecture and it differs depending on the position of an entity in the connector hierarchy (root, node, and leaf entities). Bellow, we list the operations that implement the reconfiguration process. Due to space constraints, the pseudo-code of the operations along with additional comments can be found in [5].

Lookup Port. Serves for looking up a local reference to an object implementing a particular provided port.

Lookup Remote Port. Serves for looking up a bundle of remote references to objects providing entry-points to the implementation of a particular remote server port. Each reference in the bundle is associated with an access scheme.

Bind Port. Serves for binding a particular required port to a provided port. If there is a pass-through port dependent on the port being bound, reconfiguration is initiated. Bind Port also checks for re-entrant invocations for the same ports to detect cyclic dependencies between pass-through ports.

Bind Remote Port. Serves for binding a particular remote client port to a remote server port. If there is a pass-through port dependent on the port being bound, reconfiguration is initiated.

Invalidate Port. Serves for invalidating a reference to an object implementing a particular provided port. Used by connector elements to indicate that the provided port should be queried for a new reference.

Invalidate Remote Port. Serves for invalidating a bundle of remote references to objects providing an entry point to the implementation of a particular remote server port. Used by connector elements to indicate that the remote server port should be queried for a new reference bundle.

Rebind Connector Units. Serves for establishing a remote binding – gathers reference from remote ports with server functionality (using Lookup Remote Port) and distributes the references to remote ports with client functionality (using Bind Remote Port).

Bind Component Interface. Serves for binding a particular component interface to a provided connector element port implementing the interface. Serviced by a method specific to a particular component-model.

5.2 Algorithm Termination

Given the recursive nature of the algorithm, an obvious question is whether it always terminates. We show that the answer is yes, even when an implementation of a connector architecture is invalid (i.e. there are cyclic dependencies between pass-through ports), in which case the algorithm detects the cycle and terminates. In proving that the algorithm always terminates, we will examine the reasons for the algorithm not to terminate and show that such situation cannot happen. We believe that the use of informal language does not affect the correctness of the proof.

As a requisite for the proof, we construct a call graph of the algorithm operations (see Figure 4). Implementation variants of the operations as well as invocations on different instances are not distinguished. This simplifies reasoning, because it abstracts away from unimportant details. The cycles in the graph mark the problem places that could prevent the algorithm from terminating. Even though the graph in Figure 4 does not reflect the algorithm as accurately as the more detailed graphs would, the structure of the cycles is preserved, therefore the graph is adequate for the purpose of the proof.

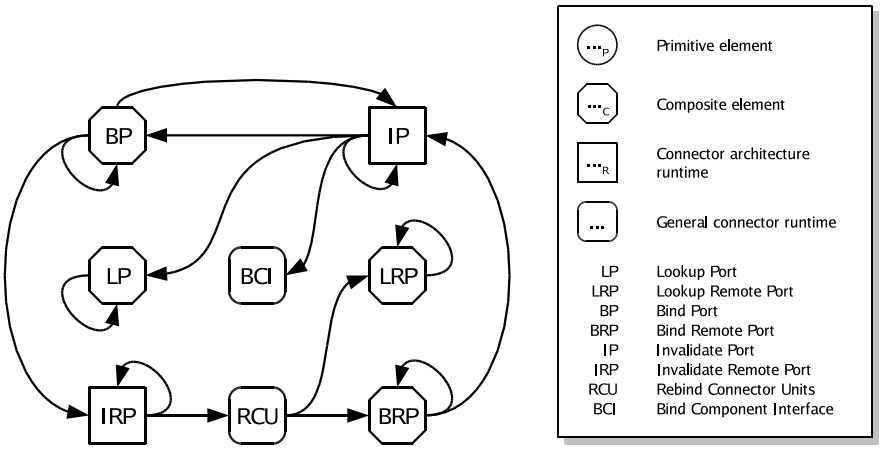

Fig. 4. Static call graph of the reconfiguration algorithm operations

The first step is to analyze the trivial cycles associated with the nodes labeled LP, LRP, BP, BRP, IP, and IRP. In case of the LP, LRP, BP, and BRP nodes, these cycles correspond to delegation of the respective Lookup Port, Lookup Remote Port, Bind Port, and Bind Remote Port operations from the parent elements to the child elements, down the connector architecture hierarchy. The recursion of these operations is limited by the finite depth of the connector architecture hierarchy – eventually the methods must reach a primitive element, which is a leaf entity of the connector architecture. In case of the IP and IRP nodes, the cycles correspond to the propagation of the Invalidate Port and Invalidate Remote Port operations from a connector element to its parent entity (another connector element or connector runtime). The recursion is again limited by the depth of the connector architecture hierarchy, but in this case these operations must eventually reach the root entity (connector runtime).

The cycles associated with the BP, BRP, IP, and IRP nodes can be also part of other, non-trivial, cycles. A class of cycles derived from the IP-BP cycle corresponds to the distribution of a new port reference to other elements. The Bind Port operation traverses the connector architecture downwards, while the Invalidate Port operation upwards. Because the number of required ports in a connector element is finite, then if there is a cyclic dependency between pass-through ports, the Bind Port operation will inevitably get called again for the same port before the previous call finished. Since the Bind Port operation has to be part of every such cycle, the operation guards against re-entrant invocation for the same port by using a per-port flag indicating that reconfiguration is already in progress for the port. The use of the flag corresponds to marking of the path during state-space traversal in order to detect cycles. Therefore if a Bind Port operation finds the flag already set in the port it was called for, there must be a cyclic dependency between pass-through ports and the algorithm terminates.

The other non-trivial class of cycles can be derived from the IP-BP-IRP-RCU-BRP cycle. These cycles can be longer and more complex than in the previous case, but they are bounded for the same reason as above – the bind port operation is guarded against re-entrant invocations and thus these cycles are also bounded by the number of required ports in a connector.

Because no other cycles are possible in the call graph, the algorithm always terminates in finite number of steps because the connector architecture is finite. In case of invalid implementation of a connector architecture, the algorithm is terminated prematurely when a cycle between pass-through ports is detected.

6 Evaluation and Related Work

The approach presented in this paper is best related to other work in the context of its intended application. Including optional features in connectors can be related to instrumenting applications with code that is not directly related to their primary function. There are various instrumentation techniques, distinguished by the moment of instrumentation, the transparency of its usage, whether the instrumentation is performed on application source or binary, etc.

Eliminating the overhead associated with disabled optional features allows us to always include these features even in production-level applications, where they only impact the execution of an application when they are used. The most prominent examples of such features are are logging, tracing, or performance and behavior monitoring. With respect to tracing, our approach can be used for dynamic tracing of component-based applications, which would provide functionality similar to that of DTrace [6]. While DTrace provides tracing mainly on the level of system and library calls, our approach is targeted at tracing at the level of design-level elements, such as components.

Similar relation can be found in performance evaluation of component-based and distributed applications, where the original applications are instrumented with code for collecting performance data. Since the code is orthogonal to the primary function of the applications, the instrumentation can take place as late as during deployment or just before execution. Such functionality is provided by the COMPAS framework [7] by Adrian Mos, which instruments an EJB application with probes that can report performance information during execution of the application. The operational status of the probes can be controlled at runtime, but even when disabled, the static overhead of the instrumentation is still present. However, since the instrumentation mechanism serves only a single purpose (it is not designed to allow combining multiple features in contrast to the compositional approach in case of connector architecture), it potentially adds only a single level of indirection and its overhead is therefore negligible.

When evaluating performance of CCA [8] applications using the TAU [9] tools for CCA, a proxy component must be generated for each component that should be included in the evaluation. The integration of these into the original application requires modification of the architecture description to redirect the original bindings to the proxy components. The proxy components, even when

inactive, still contribute certain overhead to the execution of the instrumented application. Using connectors for the same purpose would allow the instrumentation to be completely transparent to the component application, and the static overhead of the proxy components could be eliminated as well.

Similar goals, i.e. performance monitoring of distributed CORBA-based applications are pursued in [10] and in the WABASH [11] tool. The former approach uses BOA inheritance and TIE classes to add instrumentation code, while WABASH uses CORBA interceptors to achieve the same. In both cases, the instrumentation mechanism does not incur significant performance overhead, but requires source code of the application to perform the instrumentation, the availability of which cannot be always assured.

From a certain point of view, the changes performed in connector runtime could be seen as dynamic reconfiguration of middleware, and therefore be related to reflective middleware by Blair et al. [2]. However, the relation is only marginal, because the dynamic reconfiguration of middleware is a more general and consequently more difficult problem to solve. What makes our approach feasible is that the architecture of a connector in fact remains unchanged and that the reconfiguration is initiated by a connector element from inside of the architecture and only requires local information in each step.

7 Conclusion

In this paper, we have presented an approach to elimination of static execution overhead associated with disabled optional features in a particular model of architecture-based connectors. The approach is based on structural reconfiguration of runtime entities implementing a connector and utilizes runtime information on connector architecture to derive dependencies among the entities implementing a connector.

Even though the reconfiguration process which forms the basis of the approach has been presented on a specific connector model, the approach can be used in similarly structured environments, such as simple component models and componentized middleware, if the necessary information and facilities for manipulating implementation entities are available.

Apart from atomicity of a reference assignment, the reconfiguration algorithm makes no other technical assumptions and requires no operations specific to any particular programming language or platform. This makes it well suitable for use in heterogeneous execution environments.

While the reconfiguration algorithm itself would not be difficult to implement, for routine use in connectors [4] the implementation of the algorithm has to be generated. This requires implementing a generator of the algorithm implementation and integrating it with a prototype connector generator. This project is currently under way, but no case study with connectors utilizing the algorithm is available at the moment.

The algorithm undoubtedly improves the efficiency of a connector by eliminating unnecessary indirections, but the improvement has not yet been experi-

mentally evaluated and it remains to be seen how many eliminated indirections are required to witness a non-trivial improvement. In case of features such as logging or monitoring, the main achievement is that such features can be included even in production-level applications without impact on normal execution.

More promising is the application of the algorithm during initial configuration of a connector which takes place at application startup. This may result in elimination of all connector code (mainly stub and skeleton) from the call path between locally connected components.

Acknowledgement. This work was partially supported by the Academy of Sciences of the Czech Republic project 1ET400300504 and by the Ministry of Education of the Czech Republic grant MSM0021620838.

References

1. Cantrill, B.: Hidden in plain sight. ACM Queue **4**(1) (2006) 26–36
2. Blair, G.S., Coulson, G., Grace, P.: Research directions in reflective middleware: the Lancaster experience. In: Proceedings of the 3rd Workshop on Adaptive and Reflective Middleware, RM 2004, Toronto, Canada, ACM (2004) 262–267
3. Dumant, B., Horn, F., Tran, F.D., Stefani, J.B.: Jonathan: an open distributed processing environment in Java. Distributed Systems Engineering **6**(1) (1999) 3–12
4. Bures, T., Plasil, F.: Communication style driven connector configurations. In Software Engineering Research and Applications: First International Conference, SERA 2003, San Francisco, USA, LNCS 3026, Springer (2004) 102–116
5. Bulej, L., Bures, T.: Addressing static execution overhead in connectors with disabled optional features. Tech. Report 2006/6, Dept. of SW Engineering, Charles University, Prague (2006)
6. Cantrill, B., Shapiro, M.W., Leventhal, A.H.: Dynamic instrumentation of production systems. In: Proceedings of the General Track: 2004 USENIX Annual Technical Conference, 2004, Boston, USA, USENIX (2004) 15–28
7. Mos, A., Murphy, J.: COMPAS: Adaptive performance monitoring of component-based systems. In: Proceedings of the 2nd International Workshop on Remote Analysis and Measurement Software Systems, RAMSS 2004, Edinburgh, UK, IEE Press (2004) 35–40
8. Malony, A.D., Shende, S., Trebon, N., Ray, J., Armstrong, R.C., Rasmussen, C.E., Sottile, M.J.: Performance technology for parallel and distributed component software. Concurrency and Computation: Practice and Experience **17**(2-4) (2005) 117–141
9. Malony, A.D., Shende, S.: Performance technology for complex parallel and distributed systems. In: Proceedings of the 3rd Austrian-Hungarian Workshop on Distributed and Parallel Systems, DAPSYS 2000, Balatonfured, Hungary, Kluwer Academic Publishers (2000) 37–46
10. McGregor, J.D., Cho, I.H., Malloy, B.A., Curry, E.L., Hobatr, C.: Collecting metrics for CORBA-based distributed systems. Empirical Software Engineering **4**(3) (1999) 217–240
11. Sridharan, B., Mathur, A.P., Dasarathy, B.: On building non-intrusive performance instrumentation blocks for corba-based distributed systems. In: Proceedings of the 4th IEEE International Computer Performance and Dependability Symposium, IPDS 2000, Chicago, USA, IEEE Computer Society (2000) 139–143