ARCHITECTURAL RECONFIGURATION USING COORDINATED_ATOMIC_ACTIONS

ROGÉRIO DE LEMOS
COMPUTING LABORATORY
UNIVERSITY OF KENT
CANTERBURY, KENT CT2 7NF, UK
r.delemos@kent.ac.uk

ABSTRACT
The provision of services despite the presence of faults is known as fault tolerance. One of its associated activities is fault handling, which aims to prevent the reactivation of already located faults. System reconfiguration, one of the steps of fault handling, is a complex cooperative activity involving several participants, thus should be designed in a structured fashion. This position paper describes how coordinated atomic actions (CA actions) and exception handling can be applied to the architectural reconfiguration of systems.

Categories and Subject Descriptors
D.2.11 [Software Engineering]: Software Architectures – Patterns.

General Terms
Design, Reliability.

Keywords
Fault tolerance, fault handling, dynamic reconfiguration, software architecture.

1. INTRODUCTION
The current trend of building systems from existing components is compelling the usage of modelling entities at higher levels of abstraction that in the past were restricted to lower levels. One of these modelling entities is related to atomicity, which is a fundamental concept for reasoning about complex systems. An atomic transaction is a control abstraction that allows the application programmer to group a sequence of operations on objects into a logical execution unit. Atomic transactions have the properties of atomicity, consistency, isolation, and durability and can be used to ensure consistency of shared objects even in the presence of failures and concurrent access [8].

As components become more sophisticated, atomicity grows in relevance in the structuring of systems. The encapsulation of more resources enables components to delivery multiple services to several applications. In this context, atomicity can facilitate both the design and evaluation of systems when used as a mechanism for the composition of components’ services. The feasibility of such an approach at the architectural level should rely on the ability to abstract from the actual components’ behaviour. This can be achieved by using well-defined interfaces that enable to express the different roles that a component might be involved. However, for this to happen it is necessary that the traditional notion of atomicity has to be changed to a more relaxed one where, for example, the components taking part in a transaction should not be fully tied up to the whole length of the transaction [12]. Although different applications might require different forms of such quasi-atomicity, it might be the case that depending on the domain or the application being considered, different design patterns are identified. Assuming that a useful relaxed notion of atomicity could be defined and implemented, the task of incorporating this concept into a process of system development is not straightforward. For example, if the decomposition of a system inevitably leads to the identification of new behaviours, some of these failure behaviours, the transformation of a business dataflow into an implementation based on the synchronization of classes cannot be captured by a simple top-down process consisting of refinement rules. Instead, this essentially top-down process should be modified for allowing bottom-up revisions, as it is already done in the context of exception handling.

In this position paper, we investigate how coordinated atomic actions (CA actions) can be employed at the architectural level for supporting the dynamic reconfiguration of systems. CA actions are a generalized form of the basic atomic action structure [23]. They were initially devised as a structuring mechanism at the design and implementations level for enclosing multi-threaded interaction and facilitating error handling. However, instead of using CA actions for structuring an application, which would be unfeasible at a higher of abstraction because would render the unavailability of computing resources, we exploit the usage of CA actions and exception handling as structuring mechanism for providing support for isolating faults, reconfiguring systems and resuming their services. Thus fault handling can be performed in a collaborative way using exception handling within a CA action.

In this work, we assume a rather restrictive interpretation for the notion of dynamic reconfiguration, and we do not consider all the activities associated with the traditional control systems model of “sense-plan-act”. Instead of considering reconfiguration in the context of an arbitrary set of system configurations [10], this paper investigates mechanisms for supporting elementary reconfiguration operations that need to be collectively executed in
an atomic way [15], but not in any prescribed order. The order
should emerge from the nature of the operations to be executed,
and the context in which they need to be executed. This is
important for guaranteeing that the system can react to
"unexpected" situations but through "predictable" means [6].

The rest of the paper is organized as follows. In the next section,
we present a brief background on key issues that are relevant for
this paper, these are: software architectures, fault tolerance and
coordinated atomic actions (CA actions). Section 3 describes how
an architectural representation should be modified for
incorporating reconfiguration. In section 4, the approach of using
CA actions as a structuring mechanism for supporting
architectural reconfiguration is presented in more detailed.
Section 5 presents some related work. Finally, the last section
provides some concluding remarks and future directions of
research.

2. BACKGROUND

2.1 Software Architectures

The architecture of a software system is an abstraction of its
structure described as a set of connected components, their
electronically visible properties and their relationships [3].
Consequently, software architecture are usually described in
terms of its components – which represent computation units,
connectors – which encapsulate the interaction between
components, and their configuration – which characterizes the
topology of the system in terms of the interconnection of
components via connectors [16][19]. An architectural style
imposes a set of constraints on the types of components and
connectors that can be used and a pattern for their control and/or
data transfers. It restricts the set of configurations allowed [19],
and simplifies descriptions and discussions by restricting the
suitable vocabulary. Software architecture may conform to a
single style or a mix of those.

2.2 Fault Tolerance

The structure of a system is what enables it to generate its
intended behaviour from the behaviour of its components. One of
the benefits of a well-structured system is to avoid overly
complex relationships between its components, which in turn
should lead to a more dependable system [17]. Dependability is
defined as the ability of a system to deliver service that can
justifiably be trusted [1]. One of the means to attain dependability
is through fault tolerance, which aims to provide system services
despite the presence of faults. From the perspective of fault
tolerance, system structuring should ensure that the extra software
involved in error detection and error handling provides effective
means for error confinement, does not add to the complexity of
the system, and improves the overall system dependability [17].
Since the architecture of a software system is an abstraction of its
actual structure, in the following, we describe how the
architectural level reasoning might affect the incorporation of
fault tolerance into system design.

During run-time, system failure is avoided via error detection and
system recovery [1]. Error detection at the architectural level
relies on monitoring mechanisms, or probes, for detecting
erroneous states at the interfaces of architectural elements or in
the interactions between these elements. On the other hand, the
aim of system recovery is twofold. First, through error handling,
to eliminate erroneous states from the system, and second,

through fault handling, to prevent located faults from being
activated again. The main activities associated with fault handling
are: the identification of the type faults and their location, the
isolation of faulty components to avoid faults to be reactivated,
the reconfiguration of system components, and the resumption of
system services. At the architectural level, reconfiguration aims
for isolating those architectural elements that might have caused the
erroneous states.

Architectural abstractions offer a number of features that are
suitable for the provision of fault tolerance [8], including error
confinement, which is the ability of a system to avoid the
propagation of errors. They also provide a global perspective of
the system that enables high-level interpretation of system faults,
thus facilitating their identification. The separation between
computation and communication, which enforces modularisation
and information hiding, facilitates error detection, confinement,
and system recovery. The architectural configuration, being
structural constraints, helps to identify anomalies in the system
structure. The role of software architectures in error confinement
needs to be approached from two distinct angles. On one hand is
the support for fostering the creation of architectural structures
that provide error confinement, and on the other hand is the
representation and analysis of error confinement mechanisms.
Explicit system structuring facilitates the introduction of
mechanisms such as program assertions, pre- and post-conditions,
and invariants that enable the detection of potential erroneous
architectural states. Thus, having a highly cohesive system with
self-checking architectural elements is essential for error
confinement. Architectural changes, for supporting fault handling
during system recovery, can include the addition, removal, or
replacement of components and connectors, modifications to the
configuration or parameters of components and connectors, and
alterations in the component/connector network’s topology.

2.3 The Essentials of CA Actions

Transactions are a mechanism for structuring competitive
systems, which have the following properties: atomicity,
consistency, isolation and durability (also known as ACID
properties) [11]. This mechanism allows participants
(processes/threads) to access resources as if they were at their
exclusive disposal, and although transactional support allows
concurrent access, this is transparent for participants
(processes/threads). On the other hand, conversations consists of
a number of concurrent cooperating participants
(processes/threads/nodes/objects) entering and leaving
concurrently [17]. They leave the conversation synchronously
when all of them have agreed on the conversation outcome. When
an error is detected in a conversation all participants are involved
in cooperative recovery. Backward error recovery (rollback, retry,
etc.) and forward error recovery (exception handling) are allowed.
The conversation execution is invisible and indivisible for the
outside world. Conversations can be nested and when recovery is
not possible at a conversation level the responsibility for recovery
is passed to the containing conversation.

The coordinated atomic action (CA action) concept was
introduced as a unified approach to structuring complex
concurrent activities and supporting error recovery between
multiple interacting participants [23][18]. This mechanism
provides a conceptual framework for dealing with both kinds of
concurrency (cooperative and competitive [12]) by extending and
integrating two complementary concepts - transactions and conversations. Conversational support is used to control cooperative concurrence and to implement coordinated and disciplined error recovery, whilst transactional support maintains the consistency of shared resources in the presence of failures and concurrence among different CA actions competing for these resources.

Each CA action has roles, which are activated by action participants, and which cooperate within the CA action scope. Logically, the action starts when all roles have been activated (though it is an implementation decision to use either synchronous or asynchronous entry protocol) and finishes when all of them reach the action end. The action can be completed either when no error has been detected or after a successful recovery or when the recovery fails and a failure exception is propagated to the containing action. External (transactional) objects (data) can be used concurrently by several CA actions in such a way that information cannot be smuggled among these actions and that any sequence of operations on them, bracketed by the CA action start and completion, has ACID properties with respect to other sequences, in other words, actions. CA action execution looks like atomic transactions for the outside world. CA actions can be nested, so that the execution of the system can be viewed as a tree that is dynamically updated. The main rules of nesting are simple: sibling actions cannot overlap; if a participant takes part in an action, it has to take part in the father action; the action is over only if all of its nested actions are terminated.

3. ARCHITECTURAL REPRESENTATION

In order to represent the architectural representation of reconfiguration, and at the same time reduce the complexity of an architectural representation, the different services that are provided/required by an architectural element are represented through distinct interfaces [14]. In the context of this paper, we have identified the application and the configuration services interfaces [20][21]:

- Application services interface provides access to the operations associated with the implementation of business rules. The functionality of the architectural elements is observed through these interfaces.

- Configuration services interface provides access to the operations associated with the architectural reconfiguration of the system.

The clearly separation of concerns between application and configuration is important. First, the separation between application and configuration activities promotes the internal structuring of the architectural elements for constraining the access between the two parts. Second, the complexities associated with the application and configuration services are kept separate from each other, thus enabling more attention to be given to the application and its interaction with the configuration part. Third, since the operations associated with the configuration services are similar across architectural elements this promotes reuse, thus allowing a more thorough evaluation of its operations.

Figure 1 shows the internal structure of a connector that maintains a clear separation of concerns between Application and Configuration, in which each has got their own interfaces. As an instantiation of the peer-to-peer architectural style [3], the application has its provided (IP_S_Application) and required (IR_S_Application) interfaces, while configuration has an interface for requesting resources (IR_S_Resource_Request), an external interface for handling resources’ failures (IR_S_Resource_Exception), and an internal interface for handling errors from the application (IR_C_Application_Exception). The role connector Integrator is essentially to deal with the mismatches that might exist between Application and Configuration. For the case of a component, the external interfaces associated with Configuration would be provided instead of required.

![Figure 1. Internal structure of a connector.](image1)

An example of an architecture in which architectural elements support two types of interfaces is shown in Figure 2 (the notation has been slightly simplified). That diagram depicts a fault tolerant software architecture that aims to deliver reliable stock quotes from sources that are not so reliable. In this simple example, there are two sources from which stock quotes can be obtained. In case BridgeYahoo fails, the system is reconfigured for obtaining the stock quotes from BridgeLycos. The reconfiguration is coordinated by the connector Single, and the infrastructure for reconfiguration is provided by the configuration interfaces, which are kept separate from the application services.

![Figure 2. An architectural configuration with two types of interfaces.](image2)
to collaborate to deliver a specified service. Hence the role of
connectors in an architectural configuration - they embody
the description of interacting behaviour between components. It is in
this context in which CA actions should be used for supporting
the architectural reconfiguration of systems.

From the perspective of fault tolerance, the partition of the
architectural elements into application and configuration it is also
convenient because it enforces the separation of error detection
and handling from that of fault handling. While the application
part of an architectural element is responsible for the detection
and handling of errors, the configuration part is responsible for
the handling of faults. If a faulty component needs to be isolated
and the system reconfigured, then this is the responsibility of the
configuration part, after it has been notified by the corresponding
application part that an error has been detected. The structuring of
system using coordinated atomic actions (CA actions) for the
purpose of error confinement, detection and recovery has already
been investigated [23]. In this paper, we explore how CA actions
can be used in supporting the dynamic reconfiguration of systems.

4. ARCHITECTURAL RECONFIGURATION

Configuration is the process of putting components together for
the delivery of a particular service. System reconfiguration is the
process of making changes to an executing system without
requiring the system to be temporarily shutdown. There is a high
degree of variability on how dynamic architectural
reconfiguration is perceived [2]: from self-organised architectures
[10], to programmed dynamism [8]. In the context of the proposed
approach, the level of self-adaptation is not important, what is
usually required in the presence of faults is an assured
reconfiguration [20].

The process of reconfiguration can either take a system wide
view, for example, the configuration layer in the CCC system
architecture [22], or take a local view, at the level of components
and connectors, as explained in the previous section. For the latter
case, information concerning the interconnection of components
and connectors can be obtained from their architectural
configuration. In the following, we present this approach using
CA actions.

The motivation for applying CA actions to architectural
reconfiguration is based on the assumption that the connection of
two or more components/connectors is done in small steps that
can be easily undone. In such context, components/connectors can
be dynamically connected and disconnected based on the
atomicity principle. Moreover, the actions associated with the
connection and disconnection of components/connectors can be
nested, that is, they may be part of other outer actions and contain
other inner smaller actions.

For the purpose of establishing a configuration, a CA action
can be incorporated into the definition of an architectural connector.
The CA action can observe the interactions between the
components and their internal states, for either committing or
aborting a configuration of components depending on the
operational state of the components. When errors occur during
connection and disconnection of resources, these can be dealt as
internal exceptions. The specification of exceptional behaviour is
fundamental since it provides the basis for implementing the
reconfiguration strategies.

Figure 3 shows the timeline of the process of establishing a
configuration by connector Single, through the CA action
connect_Config_Single (represented by rounded rectangle). The
horizontal lines in that diagram represent roles/threats, and broken
lines mean that the thread is not active. For an application to
provide a service, it is necessary for a configuration to exist.
There are two possible configurations for this system:
Configuration_Yahoo and Configuration_Lycos. Let us
consider CA action connect_Config_Yahoo (a nested CA action of
connect_Config_Single). The FrontEnd requests Single to
establish a configuration. Single_C does that through
connect_Config_Yahoo, but for that all the participants have to
be in a stable state. The first activity inside the action is to
connect_FE, which is a nested CA action of
connect_Config_Yahoo (represented by a darker rounded
rectangle). If it is not possible to establish a configuration, for
example BridgeYahoo_C has failed (represented by the cross),
an exception is raised. Either the CA action rolls back and tries an
alternative configuration (which does not exist in this case), or
rolls forward and aborts the CA action, but before that it has to
disconnect_FE. Considering that connect_Config_Yahoo was
not successful, the operation get_quotes from application cannot
start. An alternative configuration is Configuration_Lycos, and
for establishing that configuration, the CA action
connect_Config_Lycos is invoked. Since in this case there were
no failures in the components involved in this CA action, the
actions connect_Config_Lycos and connect_Config_Single
commit. Once the configuration is established the application can
invoke get_quotes. In case BridgeYahoo_C fails, an exception
should be raised, which should be propagated to the application
level. The reason for this is that the application has to be in a
stable state before the participants can be disconnected, that is,
when disconnect_Config_Lycos commits. An alternative design
to the one described above would be to include
disconnect_Config_Lycos as part of connect_Config_Single.
However, this would create a long CA action that would last for
the duration of the application. A disadvantage of such solution is
that, resources being used by the CA action cannot be shared with
other CA actions, otherwise the design principles of CA actions
would be violated.

Figure 3. CA actions representing the provision of
configuration services.

The skeleton of the CA action connect_Config_Yahoo is shown
in Figure 4. The Interface of a CA action, identifies the
Roles which are the different threads that take part in a action,
declares the Exceptions that can be signalled to the enclosing
action, and provides the pre- and post-conditions to start and end
an action (the post-conditions for an exceptional outcome might be different). In the Body of a CA action, the nested CA actions are declared, in this case connect_FE, connect_BY, disconnect_FE and disconnect_BY, the Exceptions raised by the roles participating in the CA action, the Handlers that will attempt to bring the configuration back to a safe configuration – if successful the action will end with a normal outcome, and in case there are multiple exceptions occurring inside a CA action, they are resolved through a resolution algorithm based on an exception resolution graph declared in the Resolution part [24].

<table>
<thead>
<tr>
<th>CAAction</th>
<th>connect_Config_Yahoo;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interface</td>
<td>// components participating in the collaboration</td>
</tr>
<tr>
<td></td>
<td>FrontEnd_C, Single_C,</td>
</tr>
<tr>
<td></td>
<td>BridgeYahoo_C;</td>
</tr>
<tr>
<td>Exceptions</td>
<td>// exception signaled by the CA action</td>
</tr>
<tr>
<td></td>
<td>ConfYahoo_failure;</td>
</tr>
<tr>
<td>Precondition</td>
<td>// all the participants are stable</td>
</tr>
<tr>
<td></td>
<td>FrontEnd_C_st = stable &amp;</td>
</tr>
<tr>
<td></td>
<td>Single_C_st = stable &amp;</td>
</tr>
<tr>
<td></td>
<td>BridgeYahoo_C_st = stable;</td>
</tr>
<tr>
<td>Postcondition</td>
<td>// finishing normally all the participants are stable</td>
</tr>
<tr>
<td></td>
<td>FrontEnd_C_st = stable &amp;</td>
</tr>
<tr>
<td></td>
<td>Single_C_st = stable &amp;</td>
</tr>
<tr>
<td></td>
<td>BridgeYahoo_C_st = stable;</td>
</tr>
<tr>
<td>Body</td>
<td>Use CAAction</td>
</tr>
<tr>
<td></td>
<td>// nested CA actions</td>
</tr>
<tr>
<td></td>
<td>connect_FE, connect_BY,</td>
</tr>
<tr>
<td></td>
<td>disconnect_FE, disconnect_BY;</td>
</tr>
<tr>
<td>Exceptions</td>
<td>// internal exception</td>
</tr>
<tr>
<td></td>
<td>BridgeYahoo_failure;</td>
</tr>
<tr>
<td>Handlers</td>
<td>// exception handlers</td>
</tr>
<tr>
<td></td>
<td>BridgeYahoo_handler;</td>
</tr>
<tr>
<td>Resolution</td>
<td></td>
</tr>
<tr>
<td>End</td>
<td>Configuration_Yahoo;</td>
</tr>
</tbody>
</table>

Figure 4. CA action skeleton for Configuration_Yahoo.

A similar procedure can be applied to the change of configuration by replacing connectors. The initiative for selecting a particular collaboration is taken by a component, but it is the connector that renders the collaboration, the one that manages the configuration process. As an example, let us consider a system that aims to deliver reliable stock quotes from three Web sources that are not so reliable (a variation of [5]). The reliable stock quotes are obtained by performing majority voting on the values received from different sources. However, if one of the service providers fails (we assume crash failure), the system relies on a single source for obtaining its quotes. A simplified representation of the overall architecture is shown in Figure 5, which is composed of four components and two connectors. The architectural configuration of the Figure 5(b) is identical to that of Figure 2, except for the Voter and the BridgeMS. Figure 5(a) shows the case in which the stock quotes are obtained from three different sources for the purpose of voting. If a failure occurs, the configuration changes to that of Figure 5(b) in which one of the sources, either BridgeYahoo or BridgeLycos is used by the Single connector.

Figure 5. Architectural configuration by replacing a connector.

The timeline of the CA actions associated with the architectural reconfiguration is represented in Figure 6. The first CA action establishes a configuration for the Voter connector (connect_Config_Voter). For simplifying the diagram, the nested CA actions of connect_Config_Voter and disconnect_Config_Voter are not represented. If one of the components fails, e.g., BridgeMS_C, a new CA action has to be activated for disconnecting the configuration (disconnect_Config_Voter). Once this happens a failure exception is propagated to FrontEnd_C for the component to request a new configuration to Single_C (connect_Config_Single). The process of establishing this configuration is similar to the one previously described.

Figure 6. CA actions representing the replacement of connectors.
In order to exemplify the interaction between the application and the configuration, Figure 7 presents the sequence diagram of the process of changing a configuration by replacing connectors, as depicted in Figure 6. It shows in more detail the exchange of messages between the participating threads of CA actions that are used for structuring the provision of application and configuration services.

5. RELATED WORK

It is clear from the literature that almost all the approaches for dynamic reconfiguration rely on atomicity. However, very few actually demonstrated how this could be implemented in the context of faults and other undesirable events. As already mentioned, CA actions have been used mainly in structuring applications for the purpose of error confinement and the provision of error detection and handling [18]. The activities associated with fault handling are considered in the context of the application, and there is no explicit separation of concerns between application and configuration services. On the other hand, there are several architectural approaches that support this separation of concerns [2], in addition to the CCC system architecture [22], already mentioned.

Concerning where the reconfiguration should be managed, slightly different from the Computation, Coordination and Configuration (CCC) system architecture [22], the proposed approach does not need for an explicit configuration layer because all the configuration services are embedded in the architectural elements. However, if a wider system architectural view is needed, then the configuration layer is important for incorporating the activities associated with “sense-plan-act”, which provides the basis for the dynamic reconfiguration of systems.

6. CONCLUSIONS

There are several degrees of self-adaptability that can be associated with systems, and the provision of self-adaptability at the architectural level requires some sort of reconfiguration. These reconfigurations can either be established during design-time, thus aiming to obtain predictable behaviours, or established on-the-fly during run-time depending of the resources available. For obtaining assurances in architectural reconfiguration, it is recommended that reconfiguration should be performed through a sequence of atomic actions that would allow reaching a safe (stable and useful) state in the system configuration. There are several reasons for the provision of self-adaptability, but one of the reasons is the inevitable occurrence of faults in the system that might affect its services. If these faults are not isolated and the system reconfigured, a system failure can occur.

In this paper, we have proposed the use of coordinated atomic actions (CA actions) as a mechanism for supporting dynamic architectural reconfiguration of systems. In the context of fault tolerant computing, CA actions have shown to be an effective technique for confining, detecting and recovering from errors. This paper has shown that CA actions can be equally applied to fault handling, in particular to the activities of isolation and reconfiguration. System reconfiguration is a complex operation that is prone to faults. If these are not properly handled, the system may reach an unsuccessful and irrecoverable configuration that can lead to system failure.

Since this paper provides some preliminary thoughts concerning the application of CA actions to the dynamic reconfiguration of architectures, a lot of work remains to be done. A first task would be to perform a proper verification and validation of the ideas.
presented in the paper, and one of these issues to be investigated is whether we can obtain a complete separation between the application and the configuration for the purpose of fault handling. Another challenge would be to validate the proposed approach in the context of reconfiguration strategies that rely on the identification of resources during run-time, where resources might become unavailable due to failures.

ACKNOWLEDGMENTS
Some of the issues presented in this paper have benefited from interactions with Alexander Romanovsky and Avelino Zorzo.

REFERENCES