Research Article

Representing Tactics for Fault Recovery: A Reconfigurable, Modular, and Hierarchical Approach

Fernando J. Barros

Departamento de Engenharia Informática, Universidade de Coimbra, 3030 Coimbra, Portugal

Correspondence should be addressed to Fernando J. Barros; barros@dei.uc.pt

Received 1 December 2014; Accepted 22 April 2015

Academic Editor: Salvatore Pontarelli

Copyright © 2015 Fernando J. Barros. This is an open access article distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

We show the advantages of modular and hierarchical design in obtaining fault-tolerant software. Modularity enables the identification of faulty software units simplifying key operations, like software removal and replacement. We describe three approaches to repair faulty software based on replication, namely, Passive Replication, N-Version Replication, and Active Replication, based on modular components. We show that the key construct to represent these tactics is the ability to make ad hoc changes in software topologies. We consider hierarchical mobility as a useful operation to introduce new software units for replacing faulty ones. For illustration purposes, we use connectons, a hierarchical, modular, and self-modifying software specification formalism, and its implementation in the DESMOS framework.

1. Introduction

Replication is commonly used as the basis for enabling fault tolerance. In this technique, critical software modules are replicated, and, upon fault detection, the erroneous units are removed and service is provided by the remaining correct modules. There are several tactics for supporting replication that differ in the manner service is kept active upon a fault. We consider here Passive Replication, Active Replication [1], and N-Version programming [2] approaches. The development of fault-tolerant software depends on the ability to identify and remove the faulty code.

Modular and hierarchical software enables the development of software units with well-defined input and output interfaces [3]. Due to these characteristics, modular software units are easy to identify and, if required, to replace. In this paper we exploit the ability of modular software to be used as a framework for representing resilient software. We note that object-oriented programming does not support full modularity, as defined in this work, since objects only provide input interface missing output interfaces.

The ability to repair a network of software units is enabled by the capability to support structural changes in the software topology [4]. In particular, the basic operation required to achieve software repair is the ability to remove a faulty unit and eventually to replace it by a nonfaulty instance.

We have developed connectons [3], a modular and hierarchical approach to software development that is based on the request-reply principles of object-oriented programming. These characteristics make connectons compatible with the current approaches of object-oriented software design while improving this design with modular constructs.

Connectons support the basic primitives for replacing faulty software units and to make ad hoc changes in software topologies. In particular, connectons can represent hierarchical mobility [5], a useful construct to replace or to update software units during runtime. DESMOS, a Smalltalk implementation of connectons, provides full support for modular and hierarchical software with dynamic topology.

Common fault-tolerance techniques based on replication are described in Desmos providing the proof of concept for modular and hierarchical development of resilient software. A more general discussion on software fault tolerance and reliability can be found in [6–8].

The paper is organized as follows. Section 2 provides a formal definition of basic and ensemble connectons. Desmos, the Smalltalk implementation of connectons, is described in Section 3. Section 4 provides the realization of the Passive...
Replication tactic using connectons. Active Replication is described in Section 5. N-Version Replication using component mobility is discussed in Section 6. A comparison of these approaches and a description of related work are presented in Section 7. Conclusions and future work are given in Section 8.

2. Connectons

Connectons define two types of software units: basic and ensemble connectons. Basic connectons provide the actual method invocation, whereas envelopes are a composition of connectons and provide message passing. In component composition, basic and ensemble connectons can be used indistinctly. Connectons support a modular and hierarchical typ of software construction. Ensemble definition is stochastic since it depends on the results produced by the external connectons. These values are usually unknown, since they depend on the specific topology a connecton is part of and on the links that are established by the topology.

An output signature is a 2-tuple containing the range set of the outgoing parameters and the range set of incoming parameters. Output functions convert the set of values received by an output gate. These functions are useful when several channels are linked to an output gate and, in general, to convert values without creating special connectons.

The output function outFunctionk on output gate k of intermediate signature Dk and output signature (O_k, I_k) is expressed by

\[ \text{outFunction}_k : D_k \rightarrow I_k \]

where D_k is a sequence of values from set D_k.

Given an output gate k with output signature (O_k, I_k), we use the following definitions to simplify the specification, when the output function outFunction_k and the intermediate input signature D_k are omitted:

\[ D_k = I_k \]

\[ \text{outFunction}_k ([\varnothing | t]) = \varnothing \]

\[ \text{outFunction}_k ([h | .]) = h, \]

where \( \varnothing \) represents the absence of value and [ ] represents the empty list.

2.1. Position Connecton. To illustrate an example of a basic connecton, we employ the Position entity represented in Figure 1. This connecton has three input gates: time: \text{ax}, x, \text{state}; corresponding to actions defined in the connecton. Position has also the output gate backup: corresponding to the backup services the connecton can request to the exterior. Position receives piecewise constant acceleration values and computes the current position \( x \) by double integrating the input signal. For simplicity we describe here one-dimension positions. 2D coordinates are used in the next sections.

Position state keeps the time of the last update (time), position (x), velocity (v_x), and acceleration (a_x) values. This connecton is described by

\[ M_{\text{Position}} = \{ \text{state, time: ax, x;}, \}
\]

\[ \{(R^4,0),(R^2,0),(R, R)\}, \]

\[ R^4, (time = 0, x = 0, v_x = 0, a_x = 0), \]

\[ \{\text{action state, action time ax, action x}\}. \]
The state is set by action:
\[ \text{action}_{\text{state}}(s) \]
\[ \text{time}, x, v_x, a_x \leftarrow s \]

State variables are updated when the acceleration changes by the action:
\[ \text{action}_{\text{time,acc}}(t, a) \]
\[ \delta \leftarrow t - \text{time} \]
\[ x \leftarrow x + v_x \delta + (a_x/2)\delta^2 \]
\[ v_x \leftarrow v_x + a_x\delta \]
\[ a_x \leftarrow a \]
\[ \text{time} \leftarrow t \]
\[ \text{out backup} : (\text{time}, x, v_x, a_x) \]

This action sends the current state to the outside through gate `backup`, so it can be stored in other connectons for backup purposes. The current position at time \( t \) is computed by
\[ \text{action}_x(t) \]
\[ \delta \leftarrow t - \text{time} \]
\[ x \leftarrow x + v_x \delta + (a_x/2)\delta^2 \]

This action does not change the state variables and thus the current state is not to be saved.

2.2. Ensemble Connecton. Hierarchical composition of systems has been used as a powerful heuristic to manage complex systems. We consider that connectons can be hierarchically composed, being the resultant connecton indistinguishable from the basic connecton of the last section. This ability permits handling in a homogeneous form both basic and aggregated components. A connecton ensemble (network) is a complex connecton built by the composition of other connectons. Let \( \mathcal{E} \) be the set of names corresponding to connecton ensembles, constrained to \( \mathcal{E} \cap \mathcal{B} = \emptyset \). The model of the ensemble connecton \( \chi \in \mathcal{E} \) is defined by
\[ M_\chi = (\text{inGates}, \{\text{inSign}_g\}, \{\text{inISign}_g\}, \{\text{inFunction}_g\}, \varepsilon, M_\chi, \text{outGates}, \{\text{outSign}_k\}, \{\text{outFunction}_k\}) \]

where \( \text{inGates} \) is the set of the ensemble input gates, \( \text{inSign}_g \) is the input-output signature of every gate \( g \in \text{inGates} \), \( \text{inISign}_g \) is the intermediate signature of every input gate \( g \in \text{inGates} \), \( \text{inFunction}_g \) is the input function of every gate \( g \in \text{inGates} \), \( \varepsilon \in \mathcal{E} \) is the ensemble executive, \( M_\varepsilon \) is the model of the ensemble executive, \( \text{outGates} \) is the set of the ensemble output gates, \( \text{outSign}_k \) is the output-to-input signature of every gate \( k \in \text{outGates} \), \( \text{outISign}_k \) is the intermediate signature of every output gate \( k \in \text{outGates} \), and \( \text{outFunction}_k \) is the output function of every gate \( k \in \text{outGates} \) with \( \varepsilon \) representing the set of all names associated with ensemble executives, constrained to \( \varepsilon \cap \mathcal{B} = \varepsilon \cap \mathcal{E} = \emptyset \).

The default signatures and input/output functions defined in Section 2.1 are used to simplify the ensemble specification.

The connecton ensemble has the same type of interface of a basic connecton making it possible to use ensembles as components of other ensembles, enabling the hierarchical composition of connectons. The ensemble structure is managed by a special connecton termed here ensemble executive \( e \). The executive keeps a list of the connectons that compose the ensemble. It also keeps the set of the channels existing among connectons. This information is not static and can be changed by executive actions. The model of the ensemble executive is an augmented connecton model defined by
\[ M_e = (\text{inGates}, \{\text{inSign}_g\}, \{\text{inFunction}_g\}, \varepsilon, M_e, \text{outGates}, \{\text{outSign}_k\}, \{\text{outFunction}_k\})_\varepsilon. \]

Function \( \sigma \) maps the executive state into an ensemble structure. The structure function \( \sigma \) is expressed by
\[ \sigma : S \rightarrow \hat{\Sigma}. \]
4.1. Example: Testing Position. To illustrate the definition of an ensemble, we build a connecton to test the Position connecton of Section 2.2. The ensemble is depicted in Figure 2 and it is defined by

\[ M_{\text{PositionTest}} = ([\text{time} : \text{ax} : x], \{0, R\}, \epsilon, M_c), \]  

(10)

where

\[ M_c = ([\text{time} : \text{ax} : x :], \{0, R\}, \epsilon, \sigma, \Sigma) \].  

(11)

The ensemble has a single static structure given by

\[ \sigma(s_0) = \sigma(s_1) = \cdots = \sigma([c, M_c], L) \].  

(12)

where

\[ \begin{align*} 
 C &= \{\text{Position}\} \\
 M_c &= \{M_{\text{Position}}\} \\
 L &= \{((\text{PositionTest}, x), (\text{PositionTest}, x)), \\
 &((\text{PositionTest}, \text{time} : \text{ax} :), (\text{Position}, \text{time} : \text{ax} :)) \\
 &((\text{PositionTest}, x :), (\text{Position}, x :)) \} 
\end{align*} \]

The connecton ensemble, represented in Figure 2, is composed of one Position connecton, linked to the executive PositionTest. The ensemble has the input gate \( x \) to access the value of the current position. The executive requests the position through the call \( x : \text{time} \), where \( \text{time} \) represents the current time.

The executive updates at a regular interval the current value of acceleration \( a \), through gate \( \text{time} : \text{ax} : \). This value is integrated by connecton Position that computes current position and velocity as described in the last section. Output gate backup: is not linked and messages sent through this gate are just ignored.

2.3. Kinds of Structural Changes. Software ensembles can undergo arbitrary structural changes. These changes include the ability to add and remove connectons and the capability to modify the channels among connectons. The kinds of structural changes are illustrated with ensemble connecton \( D \) represented in Figure 3. The ensemble \( D \) is initially empty except for the executive \( D_e \), as depicted in Figure 3(a). The executive \( D_e \) creates connecton \( A \) and adds channels between the connectons \( D \) and \( A \) and between \( A \) and \( D \), as represented in Figure 3(b). The next change involves the addition of connecton \( B \), the creation of a channel from \( B \) to \( D \), the deletion of the channel from \( A \) to \( D \), and the creation of a channel from \( A \) to \( B \), as depicted in Figure 3(c). Finally, connecton \( A \) is deleted and a channel is created from \( D \) to \( B \), as represented in Figure 3(d). We note that since connectons support hierarchical software units, \( A \) and \( B \) can be either basic or ensemble connectons. Due to the reflective capabilities of the executive, structural decisions can be made taking into account the current ensemble structure. This gives the possibility of making changes based, for example, on the number of connectons or channels currently present in the ensemble. Another form of topology adaptation involves the transmission of a connecton between two ensembles [5] and is termed here hierarchical mobility.

3. The Desmos Software System

The Desmos software system provides an implementation of connectons in the Smalltalk language. Smalltalk proved to be an excellent prototyping language offering many constructs not commonly provided by mainstream object-oriented languages, for example, block closures, used in filter implementation.

3.1. Desmos Organization. In the Desmos software system, connecton models are hierarchically organized. Desmos: :Model is the root model for all connecton models. Model Executive is the base connecton model of all ensemble executives. This model implements all basic operations that support changes in connecton network structure. Operations include adding and deleting connectons and channels. Every specific domain ensemble executive must be a subclass of the Executive model. Figure 4 represents a partial view of the Desmos hierarchy, as described above. All the submodels of the mode Executive can inherit the structure of the parent model. The Executive model provides just an empty connecton ensemble and all the primitive operations that can be used to manage the ensemble structure.

3.2. Desmos Support for Structural Changes. The following methods are defined in the Desmos system to change the ensemble connecton structure during the execution of a program:

(i) add: aName model: aModel adds to the ensemble a connecton named aName and associates it with model aModel.

(ii) add: aConnecton name: aName adds aConnecton to the ensemble and names it aName.

(iii) remove: aName removes a connecton named aName from the ensemble. All channels from and to the removed connecton are removed.
Figure 3: Structural changes.

Figure 4: Desmos hierarchy.

(iv) link: aName gate: aGate to: bName gate: bGate creates a channel between two connectons.

(v) link: aName gate: aGate to: bName gate: bGate filter: dFilter filter: rFilter creates channel and filters between two connectons.

(vi) unlink: aName gate: aGate from: bName gate: bGate deletes a channel between two connectons. A connecton is referenced by its name. Names are assigned to connecton instances in the structure definition of the ensemble connecton. Connectons provide a uniform framework for defining component behavior and architectural changes. Executive methods responsible for the structural adaptations are intuitive to use due to the explicit representation of structure in the formalism.

4. Passive Replication

Fault tolerance has been subjected to intense research in the last decades and several tactics have been developed to achieve resilient software. These approaches are mainly based on replication and on the ability to detect and to remove faulty software modules. We consider first Passive Replication (PR). Other solutions are presented in next sections.

In PR, one component (the primary) handles all the communication with the service requesters. When the primary updates its state, it backs up the state variables in the passive replicas. Fault tolerance is achieved by removing the primary replica when it becomes faulty and by promoting a backup replica to play the role of the primary. PR requires the state of the primary to be stored in all backup replicas so they can be used in case of failure.

We consider a 2D variant of Position connecton described in Section 2.1 and used here in replication to achieve resilience. For illustration purposes, we use the connecton ensemble represented in Figure 5(a) and defined in Desmos by Listing 1. Fault detection uses heartbeat messages sent at regular intervals. Reverse filters are used in the ensemble definition since the executive needs not only to receive the heartbeats from connectons A, B, and C but also to associate a name to each received value. The names missing are considered to correspond to faulty connectons.

Connecton A plays the role of the primary replica and B and C play the role of backup replicas. External requests for position are sent to the executive gate xy. The executive determines the current time and sends a request for position through gate xy: that is only handled by the primary. Connecton A computes the position at the current time and returns it to the executive. When the acceleration changes, the executive sends the new value through gate time:ax:xy: so it can be stored in the passive replicas. As stated above, replicas are removed by the executive when they fail to send the heartbeat signal.

There are two distinct failure cases that need to be handled differently. A fault detected in a backup replica is handled by simply removing that replica. The removal of a faulty backup unit is defined by Listing 2. Since the removal operation also deletes all channels to and from the removed connecton, there is only a single command in this action.
The removal of the primary replica needs to be handled differently since we need to promote one of the backups to become the primary. These changes in topology are defined by Listing 3 where the new primary and the remaining replicas are sent as parameters.

Once the faulty primary is removed, the executive is linked to the new primary through gates \texttt{time:ax:ay} and \texttt{xy}. The new primary gate \texttt{backup} is then linked to the \texttt{state} gates of the remaining backup replicas.

5. Active Replication

We consider now the Active Replication tactic to achieve fault tolerance. In this approach, several components are simultaneously active and can give an answer to any request. However, only the first answer, in the case of nonmalicious replicas, is taken into account, the others being ignored [9]. Contrarily to the Passive Replication, all replicas perform the same computations, this approach becoming more CPU

\textbf{Figure 5: Passive Replication topology management.}
intensive. This strategy is heavily dependent on the ability to establish group communication in a deterministic manner, since results are dependent on the order requests are made [9]. Connections select function can be used to establish the order of group communication. This function is implemented implicitly in Desmos that follows the order in which channels are declared in the structure definition.

The ensemble of Figure 6(a) represents an initial software topology for the Active Replication. Connectors A, B, and C play the same role in the active approach. Requests for position are sent to all the connectors in a deterministic order. This approach requires the asynchronous handling of replies since only one answer is required, since we consider here only the case of nonmalicious replicas. Thus, connectors do not give a direct answer but only answer an (id)entifier that is used to match callbacks that are sent through gate reply:id:. Callbacks are handled asynchronously, and the first reply to arrive to the executive is considered to be the answer. Later callbacks are ignored.

Fault detection is based on heartbeat, like in the previous section. Since all connectors are identical, their removal is described by Listing 4, since no further distinction is required.

The software topology becomes represented by Figure 6(b) after the removal of the faulty connector B.

6. N-Version Replication

We consider now N-Version as a means of achieving resilience. N-Version uses different versions of software units with the same functional requirements [2] to obtain N-results for the same call. These results are then compared, and modules that have produced values considered to be wrong are treated as faulty and removed. A topology for N-Version is represented in Figure 7(a). The executive broadcasts the request for position to connectors A, B, and C and waits for all the answers.

We consider hierarchical mobility [5], as a useful construct to replace faulty units. The Executive obtains the current position from all the PositionNV software units and tests if the returned values are within some tolerance limit. In case there are discrepancies, it finds and removes the faulty unit and sends it through gate fault:. A new unit arrives through gate update:name: bringing a replacement. Actually, any connector compatible with the PositionNV model can be received. Given the hierarchical nature of connectors, an ensemble connector can be used to replace a basic connecton, or vice versa. If we take, for example, software unit B as faulty, the ensemble becomes represented by Figure 7(b) after the removal of B. The removal of a faulty unit is defined by Listing 5.

The remove operation (Line 3) returns a software unit that can be handled as a regular object. This data is then sent through output gate fault: (Line 4), enabling hierarchical mobility. When a PositionNV replacement unit arrives, it is linked to the executive LocationNV by the action update:name: defined in Listing 6.

The executive LocationNV uses its output gate state to retrieve the current state of the remaining connectors so the arriving one can be initialized. Since state information has a fixed format, it becomes crucial that the replacing mobile connecton understands that format. This is easy to achieve since state is known at the design time of the (mobile) connectors used to support fault tolerance. The ensemble of Figure 7(c) represents the correction achieved by PositionNV::K used to replace the faulty connecton PositionNV::B previously removed.

As shown in this example, updates can use hierarchical mobility to introduce new versions of the software units. These updates do not require the detection of faults and can be motivated by the release of a more recent version of software. Hierarchical mobility provides, thus, a unifying
construct to represent both software update and fault handling.

7. Comparison of the Replication Approaches and Related Work

We have presented a possible realization of three common replication approaches to fault-tolerance software. For simplicity, only one fault detection method was used in each tactic. Most common methods include exception handling, heartbeat (considered here), voting (used in the N-Version), and ping/echo [10]. These methods can be merged, and different implementations can require handling them in combination. We have considered the executive to perform most of the work. However, executive, functionally, can be broken in separate connections. Some of the error handling could be used in a separate connection, for example, so it could be reused in the different tactics.
Figure 7: N-Version software topology management.
We do not foresee, however, advantages in the systematic partition of components into normal and abnormal subcomponents [11, 12], since this decision seems to be domain dependent. In our case, faults (represented by the absence of the heartbeat message) are signaled to the executive that controls the structure, and no local treatment is required. In case of exceptions that can be handled locally, we consider that local methods can be a better choice, since they can use the local state in error correction. If handled by the abnormal component, faults would require the transmission of the local state. Requests dealing with faults that cannot be treated locally can be sent through an output gate to any external component that can handle it. The requirement for a mandatory abnormal component seems thus excessive.

We consider connectons as units of software reuse, and unless we have the evidence that a particular abnormal connecton can be used in different contexts, there is mostly no point in developing it. Tactics can be used in combination taking advantage of hierarchical composition. A unit used in N-Version can be implemented, for example, using the Active Replication approach. Since ensembles hide their internal structure, it will be transparent for the N-Version approach in which some or all of the units are implemented as a combination of connectons developed using different tactics.

The initial Position connecton described in Section 2.1 has suffered several changes so it could be used by the different tactics and fault detection mechanisms. This situation seems to indicate that it may be difficult to keep models and fault tolerance orthogonal. In our case, the nonfunctional requirement of resilience has become part of the models. The use of structural inheritance [3] can be employed to minimize the impact of transforming a base model into the several versions required by the different tactics for fault-tolerance detection and recovery.

Hierarchical mobility can be used as an effective construct to replace software units or to make their update in any of the tactics described. This can be particularly useful for preventing future crashes. Bugs can be detected in some of the currently running components. After correction, these new versions can be put into production before they manifest in many other systems. This strategy is nowadays common and is used, for example, in the online update of antivirus and operating systems. Hierarchical mobility, however, provides a finer grain control over the components that need to be replaced.

Hierarchical and modular principles have been used in many fields as a powerful heuristic for handling complex problems. One of the first formal descriptions of modular decomposition has been made in the area of General Systems Theory [13]. The decomposition of software in modules has latter been advocated in software engineering [14]. In this work, however, the hierarchical decomposition of software has not been introduced and the term hierarchy is simply used as synonymous of layered (software). We have extended General Systems Theory with dynamic topologies [15]. This work, however, could not be directly applied to software engineering since it is based on general systems asynchronous and unidirectional messages, making the specifications of software systems cumbersome.

Software architectures have been developed to overcome the limitations of the object-oriented paradigm. Software components, as opposed to software objects, are intended to be built independently from the interconnections they may be part of, enabling a stronger form of reuse. A large variety of architecture definition languages have been developed but many are façades providing little support for developing a complete implementation of components and their interconnections [16–18].

We have developed a system able to represent dynamic structure hierarchical and modular software that is fully compatible with the object-oriented style [3]. Other executable software architectures have also been proposed [19, 20], but they exhibit strong limitations, which makes these approaches incapable of providing the solutions presented here for replacing faulty software. In particular, ArchJava [19] requires an exact match of the gates so they can be connected. Additionally, links in ArchJava provide no filtering capabilities. ArchJava has limited capabilities to change software structure featuring no explicit operators to remove components or links. To the best of our knowledge, connecton is the only framework supporting hierarchical mobility [21–23].

8. Conclusions and Future Work

In this paper we propose an approach to fault-tolerant software based on modular software units. Connecton and its
implementation in Desmos support the basic operators that enable dynamic changes in software topology. Most common fault-tolerant software approaches involve replication and the ability to remove faulty software. These basic constructs map easily into software topologies that, like connections, support ad hoc changes in their structure. In particular, we have shown that Passive Replication, Active Replication, and N-Version programming can be easily modeled as dynamic structure software topologies. Hierarchical mobility has also been shown as an effective construct to update faulty software modules during runtime. Connections are currently supported in a Java version [24]. Connections/Java will be employed in the development of fault-tolerant software applications for real-world systems. Future work will also address the representation of the Active Replication approach in the presence of malicious replicas.

Conflict of Interests

The author declares that there is no conflict of interests regarding the publication of this paper.

References


Submit your manuscripts at
http://www.hindawi.com