Robust Reconfigurations of Component Assemblies

Fabienne Boyer
LIG
Université Joseph Fourier
Grenoble, France
Fabienne.Boyer@imag.fr

Olivier Gruber
LIG
Université Joseph Fourier
Grenoble, France
Olivier.Gruber@imag.fr

Damien Pous
CNRS
Grenoble, France
Damien.Pous@ens-lyon.fr

***Abstract*—In this paper, we propose a reconfiguration protocol that can handle any number of failures during a reconfiguration, always producing an architecturally-consistent assembly of components that can be safely introspected and further reconfigured. Our protocol is based on the concept of Incrementally Consistent Sequences (ICS), ensuring that any reconfiguration incrementally respects the reconfiguration contract given to component developers: reconfiguration grammar and architectural invariants. We also propose two recovery policies, one rolls back the failed reconfiguration and the other rolls it forward, both going as far as possible, failure permitting. We specified and proved the reconfiguration contract, the protocol, and recovery policies in Coq.**

***Index Terms*—Dynamic reconfiguration, Component models, Robustness**

I. Introduction

A current trend in dynamically reconfigurable systems is the use of a model-driven approach to govern the evolution and maintenance of component assemblies [3][5][11][20]. In this approach, an administrator (or an autonomic tool) is provided with a component-based model of the software architecture of the complex system he manages. As needed, he can introspect that model to analyze the current architecture and shape a desired software architecture. Given as input these two architectures, a reconfiguration protocol is responsible for driving the effective evolution of the managed system from its current architecture towards the desired target architecture.

Traditionally, a reconfiguration protocol is based on an *architectural diff* that compares the current and target architectures, producing a set of elementary reconfiguration operations to apply on software components [21][3][13][15][23]. Through this apply process, the reconfiguration protocol evolves the component assembly from its current architecture towards the desired one, one reconfiguration operation at a time. In [21], the authors discuss the crucial role of this protocol: a safe ordering of reconfiguration operations.

A safe ordering must respect the reconfiguration contract under which components have been programmed. This contract classically defines a reconfiguration grammar specifying legal sequences of reconfiguration operations per component. With this knowledge, developers design each component as a finite state machine that reacts to the reconfiguration operations issued by the reconfiguration protocol. As briefly described in [21], the traditionally-accepted ordering is the one derived from the reconfiguration protocol originally introduced in [17].

We argue that this traditional ordering suffers from two main limitations today. Firstly, component models have evolved since then, introducing different semantics on component dependencies, optional and mandatory [8][9][10][25], which controls if components are started or stopped. This evolution of the component paradigm induces an evolution of the programming contract given to component developers, extending the reconfiguration grammar and introducing architectural invariants. While we suspect that the impacts on the ordering of reconfiguration operations has been considered by modern component-based systems, to the best of our knowledge, these impacts have never been discussed in any published materials.

Secondly, the traditional ordering does not support failures occurring at reconfiguration time. A component may indeed fail to successfully apply a reconfiguration operation. To face this limitation, [13][14][19] advocate a rollback strategy based on inverse operations. However, this approach only tolerates a single failure since the rollback only succeeds if all inverse operations succeed. Given that inverse operations are regular reconfiguration operations, there is no guarantee that they will succeed. In case an inverse operation fails, the rollback must be interrupted, leaving human administrators or autonomic managers with the difficult task of introspecting and repairing a partially rollbacked and potentially inconsistent assembly.

The contributions of our work are the following. First, we propose the first reconfiguration protocol that orders reconfiguration operations such as to respect the reconfiguration contract given to component developers, fulling specifying both the reconfiguration grammar and the architectural invariants that must be respected. Second, our protocol is robust, respecting the reconfiguration contract even in the case of multiple failures occurring during a reconfiguration. Third, we formalized the complete protocol and proved it using the Coq proof assistant [4], providing one of the strongest degree of guarantee currently available with formal methods.

The core principle behind our solution is that the reconfiguration protocol always evolves the component assembly from one consistent architecture to another consistent architecture, only through a path of *architecturally consistent* architectures. This approach permits to manage failures as regular reconfigurations: when a component fails to execute a requested operation, our protocol stops, marks the component as failed, and

then propagates the effects of that failure throughout the component assembly. Since this propagation is a reconfiguration, it may induce further failures that are processed identically. Once the impacts of all failures have been propagated, the managed system is architecturally consistent, which means that it can be safely introspected and further reconfigured. Importantly, this means that an administrator or an autonomic tool can safely introspect this architecture and decide how to best repair the occurred failures.

The remaining of the paper is structured as follows. Section II discusses component models. In Sections III and IV, we detail our reconfiguration protocol in the absence of failures, while Section V is devoted to the robust version of the protocol. Section VI evaluates the proposed protocol. Finally we discuss related works in Section VII and we conclude in Section VIII.

II. Component Model

This section briefly recalls the growing consensus amongst component models [8][9][10][25] regarding the concept of a component, its lifecycle, and its dependencies, as well as the reconfigurability of a component assembly.

A component is a software entity that defines a set of exports and imports. Exports describe services that the component is willing to provide while imports describe services that it requires to function properly. Hence, a component assembly can be shaped by wiring imports to exports. Imports are given either a mandatory or optional semantics; while optional imports may be wired or unwired at any time during the lifetime of a component, mandatory imports should be wired before a component is started. In other words, to be fully functional, a component must have all its mandatory imports wired to exports.

The main reconfiguration operations that are provided are the following.

Reconfiguration operations

- CONSTRUCT/DESTRUCT components
- WIRE/UNWIRE components
- START/STOP components

Each component implements all these reconfiguration operations in a component-specific manner but following a *reconfiguration contract* including a *reconfiguration grammar* and *architectural invariants*. The reconfiguration grammar, as depicted in Figure 1, specifies in which order and in which conditions a reconfiguration operation may be invoked. This grammar relies on three main lifecycle states for a component: STOPPED, STARTED, and FAILED. When first constructed, a component is in the STOPPED state, meaning it is not functional yet. A component may reach the STARTED state once it is resolved, i.e., once all its mandatory imports are wired. A component may fail at any time, reaching the FAILED state. Once failed, a component may be removed from the assembly.

Architectural invariants, defined below, constrain the architecture of a component assembly that corresponds to the description of which component compose the assembly, what are their lifecycle states and how they are wired together. More preciselly, architectural invariants correlate the components lifecycle with the semantics of their imports (optional vs mandatory). These invariants define an *architecturally consistent* assembly of components, meaning an assembly that can be safely introspected and reconfigured by an administrator or an autonomic tool. They are not to be confused with higher-level invariants that capture application-specific and domain-specific knowledge. For instance, application-specific invariants might state that an application only works if all components are started, or that certain components may be stopped and the application is still operational. Hence, an architecturally-consistent assembly might not be functional from an overall application perspective, but it is reconfigurable in order to re-establish a functional assembly.

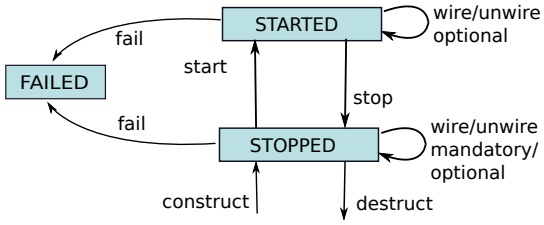

Fig. 1. Reconfiguration grammar

Definition 1. Architectural invariants

(I_1) *All started components have all their mandatory imports wired.*

(I_2) *All started components are wired only to started components.*

(I_3) *There are no wires to/from failed or destructed components*

(I_4) *There are no cycles through mandatory imports*[1].

The reconfiguration grammar along with the architectural invariants define the reconfiguration contract that enables component developers to approach the design of a component as a Finite State Machine. For instance, once started, a component has the guarantee that all its mandatory imports have been wired. Reversely, any component will be stopped before its mandatory imports are unwired. This contract is therefore the cornerstone of component design, helping developers to master the difficult challenge of designing components that can be dynamically reconfigured.

Notation

In the sequel, we use letters c, d to range over components, and v, w to range over wires. A wire w links an import i of a component c to an export e of another component c'; we denote the $source(c)$ by $w.src$ and the $destination(c')$ by $w.dst$. We also use letter o to range over *reconfiguration operations*, which we separate into *down operations:* $stop(c)$,

[1]This is a widely-accepted limitation amongst component models [8][9][10][25].

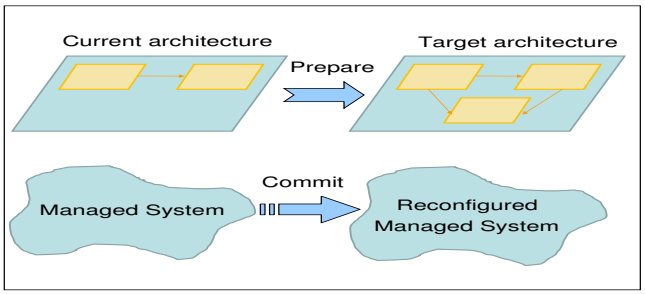

Fig. 2. Reconfiguration session

unwired$(w) \Rightarrow$
  if *mandatory*(w) ***stopped***$(w.src)$
  unwire(w)

stopped$(c) \Rightarrow$
  for all w such that $w.dst = c$
    if *mandatory*(w) ***stopped***$(w.src)$
    else **unwired**(w)
  stop(c)

destructed$(c) \Rightarrow$
  for all w such that $w.src = c$ or $w.dst = c$
    unwired(w)
  destruct(c)

Fig. 3. Propagation rules

$unwire(w)$, $destruct(c)$, $fail(c)$, and *up operations*: $construct(c)$, $wire(i, e)$, $start(c)$.

III. Reconfiguration Challenge

A reconfiguration protocol [3][13][15][21][23] reconfigures a component assembly according to a desired *target architecture* (specifiying which components should compose the system, what are their lifecycle states, and how they should be wired together according to Section II). Each reconfiguration is organized as a session with two phases, as depicted in Figure 2. In the *prepare* phase, a working copy of the current architecture can be freely reshaped to define the *target architecture*[2]. Then, during the *commit* phase, the reconfiguration protocol evolves the managed system towards the target architecture by applying reconfiguration operations.

The challenge of a reconfiguration protocol consists in finding a correct sequence of reconfiguration operations. Indeed, a naive approach consisting in computing an *architectural diff* [17][21] and directly applying the corresponding reconfiguration operations may violate the reconfiguration contract. A very simple example illustrates this.

Let's suppose that we have a simple assembly with three started components c, c_1, and c_2, with a mandatory wire w from c to c_1. Let's further suppose that the administrator gives a target architecture where the three components are also started, but where w now points to c_2. The diff will produce only two operations: $unwire(w)$ and $wire(c, c_2)$. While the architecture is consistent before and after the reconfiguration, it goes through an inconsistent state: as soon as we apply the $unwire$ operation, we violate invariant (I_1) since we have a started component (c) with an unwired mandatory import.

In order to always respect the reconfiguration contract, the managed system should always be reconfigured through incrementally consistent sequences:

Definition 2. *Given an architecture A, an* **incrementally consistent sequence (ICS)** *is a sequence $o_1, \ldots, o_n$ of reconfiguration operations such that:*

1) *for all $i \in [0, n]$, the architecture A_i obtained by successively applying the i first operations $o_1, \ldots, o_i$ to A is consistent (Definition 1);*
2) *for all $i \in [1, n]$, the operation o_i is allowed in the state A_i by the reconfiguration grammar (Figure 1).*

Intuitively, a sequence is incrementally consistent if it can be applied incrementally to a system without ever violating the reconfiguration contract: architectural invariants (Definition 1) and reconfiguration grammar (Figure 1). Note that incrementally consistent sequences can be composed to build larger sequences that remain incrementally consistent. The next section details how our reconfiguration protocol computes and applies incrementally consistent sequences (ICSs).

IV. Reconfiguration Protocol

When a reconfiguration session commits, using the current and target architectures as inputs, the reconfiguration protocol computes and applies an ICS that evolves the managed system into the target architecture. To compute such an ICS, two phases are necessary.

The first phase focuses on down operations, computing a first architectural diff between the current and target architectures, called the *Apply Down Set* (ADS) (see Listing 1). The ADS only contains down reconfiguration operations and may be incomplete, as explained earlier. Hence, rather than considering the ADS as a set of reconfiguration operations that should be applied, the ADS is considered as a set of *reconfiguration goals* that should be reached. For instance, if we have an operation $unwire(w)$ in the ADS, we will consider that we have a goal $unwired(w)$. Similarly, if we have an operation $stop(c)$, we will consider that we have a goal $stopped(c)$.

The difference between a reconfiguration operation and its corresponding reconfiguration goal is that a reconfiguration operation is a standalone operation whereas a goal may require other goals to be reached in order to preserve the architectural invariants. For instance, in the previous example, the goal $unwired(w)$ requires the goal $stopped(w.src)$ to be reached to preserve the invariant (I_1).

Our protocol uses the propagation rules given in Figure 3 to saturate the ADS with the missing reconfiguration opera-

[2]Note that the target architecture may also be selected from a version space or generated through model-driven generative approaches.

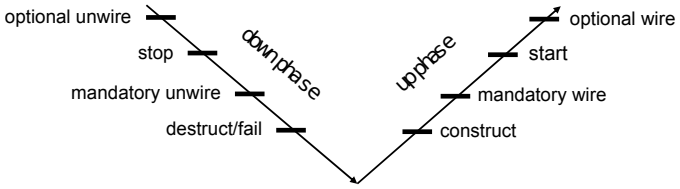

Fig. 4. Ordering to obtain an ICS

tions, producing the Saturated Apply Down Set (SADS). The first rule translates an $unwired$ goal into the corresponding $unwire$ operation but also generates a $stopped$ goal if the wire to be removed is mandatory—thus enforcing the invariant (I_1). The second rule translates a $stopped$ goal on a component c into the corresponding $stop$ operation but also generates additional goals to preserve the invariant (I_2). The third rule translates a $destructed$ goal into the corresponding $destruct$ operation but also generates $unwired$ goals for all wires connected to or from the destructed component—thus preserving the invariant (I_3). Note that this propagation always terminates; the only case where the algorithm could loop is when trying to stop a component belonging to a cycle of mandatory wires, which is precisely forbidden by the invariant (I_4).

Then our protocol obtains a first ICS by ordering the operations in the SADS such as to respect the reconfiguration grammar, as depicted in Figure 4. Furthermore, the stop operations are also ordered as to respect the invariant (I_2). Applying this ICS on the managed system ends the first down phase of the commit. Note that to apply an ICS on the managed system, the necessary quiescence must be established at the level of the component runtime [17].

The second phase of the commit is concerned with up operations. To obtain the *Apply Up Set* (AUS), it is necessary to process an up diff between the current architecture that has just evolved through down operations, and the unchanged target architecture. Note that this up diff may only be computed after the down diff has been computed *and saturated*, as the saturation may forces additionnal down operations to be processed, which will require a larger set of up operations to reach the target architecture.

The AUS obtained from the up diff is saturated by definition since architectural invariants do not require any propagation regarding up operations. This AUS needs to be ordered into an ICS, using the ordering algorithm (Figure 4). Furthermore, as it was the case for stop operations, start operations are also ordered as to respect invariant (I_2). Applying this ICS ends the second up phase of the commit.

Putting it all together, the two phases of the commit are summarized below, explaining the algorithm given in Listing 1:

1) **Down Phase.** Our protocol first computes the ADS by processing a diff between the current (A_C) and target (A_T) architectures, saturates the ADS into the SADS, orders the SADS into an ICS, and finally applies that ICS on the managed system, which evolves it to a new current architecture A'_C.
2) **Up Phase.** Our protocol computes the AUS by processing a diff between A'_C and A_T, orders this AUS into an ICS, and applies this ICS on the managed system with the architecture A'_C, which evolves it to a new current architecture A''_C.

```
commit(A_C,A_T) {
  // Reconfigure the architecture (A_C) of the managed system
  // to match the target architecture (A_T)
  assert(consistent(A_C) && consistent(A_T));

  ADS, AUS: sets of reconfiguration goals
  SADS, SAUS: sets of reconfiguration operations
  ICS: sequence of reconfiguration operations

  // down phase
  ADS = diff_down(A_C,A_T);
  SADS = propagate(A_C, ADS); // see propagation rules
  ICS = order(A_C,SADS); // see Figure 4
  A'_C = apply(A_C,ICS);

  // up phase
  AUS = diff_up(A'_C,A_T);
  ICS = order(A_T,AUS); // see Figure 4
  A''_C = apply(A'_C,ICS);

  assert(isomorph(A''_C,A_T));
  return (A''_C);
}
```

Listing 1. Commit algorithm

At the end of the commit, the architecture of the managed system (A''_C) is isomorphic to the desired target architecture (A_T), and the system has been reconfigured through two ICS.

We conclude this section with an illustrative example. Consider the managed system with the current architecture A_C from Figure 5, with four started components d, c, c_1, and c_2, mandatory wires w from d to c and w_1 from c to c_1, and an optional wire v from c_1 to c (invariant (I_4) holds since it concerns only mandatory wires). Take A_T as target architecture, where the wire w_1 is now replaced by a mandatory wire w_2 from c to c_2.

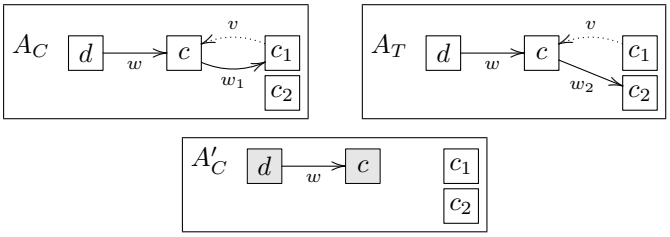

Fig. 5. Example of a reconfiguration session

The Apply Down Set returned by the first architectural diff is just the singleton $\{unwired(w_1)\}$. By the above propagation rules, we get the following saturated set of down operations: $SADS = \{unwire(w_1), unwire(w), stop(c), stop(d)\}$. This set is ordered into the following ICS: $ICS = unwire(v), stop(d), stop(c), unwire(w_1)$. Note that the component d has to be stopped first because of its mandatory dependency on c. By applying this sequence on the managed system, we get an architecture A'_C where c, d are stopped, and w is the only wire (see Figure 5). We can finally engage in the up phase.

The second architectural diff between A'_C and A_T gives the set: $AUS = \{wire(w_2), wire(v), start(c), start(d)\}$, which is ordered as follows into the following ICS: $ICS = wire(w_2), start(c), start(d), wire(v)$. As expected, applying this ICS on A'_C results in a managed system which is isomorphic to A_T.

V. Robust Protocol

During the commit phase of a reconfiguration session, failures may occur since reconfiguration operations are invoked on components and components may fail executing these reconfiguration operations. We assume that a reconfiguration operation that fails when invoked on a component c only impacted that component c. This section discusses how our protocol recovers from such failures.

When a reconfiguration operation fails, the reconfiguration protocol suspends the on-going reconfiguration and starts a recovery process. This recovery process must not only mark the component that just failed as FAILED, but it must also propagate the impacts of that failure throughout the managed system.

```
robust_commit(A_C,A_T) {
  // Reconfigure the architecture (A_C) of the managed system
  // to match the target architecture (A_T)
  assert(consistent(A_C) && consistent(A_T));

  [A'_C,FailedComp] = commit(A_C,A_T);
  if (FailedComp == null) return [A'_C,∅];
  else return recover_current(A'_C,{FailedComp});
}

recover_current(A_C,FailedSet) {
  // account for failed components in the managed system
  SADS = propagate(A_C, FailedSet);
  ICS = order(A_C,SADS);
  [A'_C,FailedComp] = apply(A_C,ICS);
  if (FailedComp == null)
    return [A'_C,FailedSet];
  else
    return recover_current(A'_C,FailedSet ∪ {FailedComp});
}
```

Listing 2. Robust commit algorithm

The corresponding **robust_commit** algorithm, given in Listing 2, wraps the regular **commit**(A_C,A_T) algorithm given earlier in Listing 1. When a failure occurs, the regular commit suspends itself and returns the identification of the component that just failed. The robust commit enters its recovery process (**recover_current**), starting with a (FailedSet) that only contains the component that just failed. It computes a saturated ADS (SADS) for accounting this failure in the managed system, using the three previous propagation rules (Figure 3) and the following one for failed goals:

failed$(c) \Rightarrow$
  for all w such that $w.src = c$ or $w.dst = c$
    unwired(w)
  fail(c)

This SADS is then ordered into an ICS by ordering the operations as depicted in Figure 4. Overall, this ICS propagates the impacts of the failure throughout the managed system and terminates by marking the component as FAILED. Since applying this ICS invokes reconfiguration operations on components, cascading failures may occur. Handling and recovering such failures is no special case. The on-going recovery suspends itself, the component that just failed is added to the set of failed components, and a new recovery is attempted. This fixpoint always terminates since the number of components that may fail is finite. When this recovery eventually completes, the managed system is architecturally consistent.

From the perspective of the managed system, the entire recovery phase appears as a sequence of ICSs, all interrupted by new failures but for the last one that manages to complete the recovery of a consistent assembly.

VI. Evaluation

We evaluated our reconfiguration protocol along three main aspects. First, we validated that our protocol always behaves as expected. Second, we evaluated the usability of our ICS-based approach to support advanced recovery policies. Third, we evaluated the scalability of our protocol.

A. Proving the Protocol

To validate that our protocol always does what it is supposed to do, we formalized and proved the entire protocol using the Coq proof assistant [4]. Coq is an *interactive theorem prover*, in the sense that it can be used to certify mathematical proofs, using an interactive process where the user progressively writes the proof with the help of the assistant. Unlike automatic theorem provers or testing-based approaches, the user has to provide a complete proof, a complex task that makes it possible to consider arbitrarily complex problems, providing one of the strongest degree of guarantee currently available with formal methods. In particular, we proved that all our algorithms behave correctly in all possible cases, including the ones that could not be tested due to material restrictions (memory or time).

The Coq development consists of about 2000 lines of Coq code that can be browsed online [1]. Given two finite sets of `Components` and `Imports`, an architecture is modeled by a record:

```
Record arch := {
  status: Component → Status;
  wires: Component*Imports → option Component
}
```

where `status` is a function indicating whether a given component is started, stopped, or failed, and `wires` is a function describing how the imports of a component are wired to other components. Reconfiguration operations are represented using an union type `op`, and we define the meaning of these operations using a function `apply`: `arch*op` → `arch`. Accordingly, we model the propagation algorithm as a function `propagate`: `arch*op` → `list op`, and the architectural diff algorithm as a function of type `arch*arch` → `list op`. Using these functions together with auxiliary functions for sorting lists of operations, we define a `commit` function, that takes a current architecture and a target architecture and produces a list of reconfiguration operations to be applied.

To model failures, we assume an arbitrary function:

```
eapply:
  arch*op*list Component → arch + Component
```

which applies a given operation to an architecture (knowing that the components given by `list Component` are failed) and can either return the new architecture, if the operation could successfully be applied, or return a newly failed component. We then define the `robust_commit` function as a fixpoint which uses the above `eapply` function to simulate an execution where an arbitrary number of failures may occur.

While we could state the various properties that should be ensured by the protocol using only 14 theorems, 153 intermediate lemmas were required to prove them. Among the high-level theorems, we proved that:

1) the commit algorithm always produces ICS, which means by definition of an ICS that the architectural invariants and the reconfiguration grammar are never violated;
2) without failures, the commit algorithm always evolves the source architecture to the target architecture;
3) the robust-commit algorithm properly acknowledges failed components, whatever the number of failures;
4) all algorithms terminate.

Among the auxiliary lemmas, we had to prove that: 1. the propagation algorithm produces saturated sets of reconfiguration operations; 2. the above saturated sets can always be ordered into an ICS; 3. the architecture reached after the down phase makes it possible to reach the target architecture using up operations only; 4. the diff algorithms are complete.

B. Leveraging our Protocol

We leverage our protocol to design two recovery policies—*Roll-Forward Policy* (RFP) and *Roll-Backward Policy* (RBP), see Listing 3—that both exploit the incremental consistency of our approach. In case of failures, the RFP policy is an automated attempt to pursue the reconfiguration as far as possible towards the desired target, failure permitting. From our previous work on repair management [24], it appeared that doing as much as possible of the originally intended reconfiguration may help the administrator to analyze and understand the failure and its impacts. Conversely, the RBP policy is an automated attempt to rollback the reconfiguration session as much as possible, failure permitting as well.

```
// Policies for reconfiguring the architecture (A_C) of the
// managed system

RFP_commit(A_C,A_T) { // Roll-Forward Policy
  [A'_C,FailedSet] = robust_commit(A_C,A_T);
  if (FailedSet == ∅) return A'_C;
  A'_T = recover_target(A_T,FailedSet); // recover reachability
  return RFP_commit(A'_C,T');
}

RBP_commit(A_C,A_T) { // Roll-Backward Policy
  [A'_C,FailedSet] = robust_commit(A_C,A_T);
  if (FailedSet == ∅) return A'_C;
  A'_T = recover_target(A_C,FailedSet); // recover reachability
  return RFP_commit(A'_C,A_I);
}

recover_target(A_T,FailedSet) {
  // account for failed components in target arch. (A_T)
  SDAS = propagate(A_T, FailedSet)
  ICS = order(A_T,SADS);
  [A'_T,_] = apply(A_T,ICS); // cannot fail
  return A'_T;
}
```

Listing 3. Roll-Forward and Roll-Backward commit algorithms

In fact, these two policies differ in their choice of the target architecture after a failure. For the Roll-Forward policy, we keep the original target architecture (A_T in the *RFP_commit* algorithm). For the Roll-Backward policy, we toss away the original target architecture and replace it with the initial architecture A_I that the managed system had before it was reconfigured.

Whichever is the new target architecture, it must be revised to incorporate the failures that occurred. This is performed by the (*recover_target*) algorithm given in Listing 3. Please note that, since the target architecture is purely abstract, reconfiguring it cannot induce cascading failures. With a target architecture that is now reachable, our protocol can resume its normal processing, pushing towards the chosen target architecture, either roll-backing or roll-forwarding.

Notice that the Roll-Forward algorithm is a fixpoint, pushing towards the chosen target as far as possible. Note also that the Roll-Backward algorithm switches to the Roll-Forward algorithm upon the occurrence of the first failure. The rationale is that, despite new failures, the commit should keep pushing towards the initial architecture of the system.

C. Case Study

In this section, we illustrate the use of our protocol to manage a Web application server whose architecture is depicted in Figure 6(a). In the figure, A_1 and A_2 represent an HTTP Daemon, T_1 and T_2 represent a Servlet Engine, and DB_1

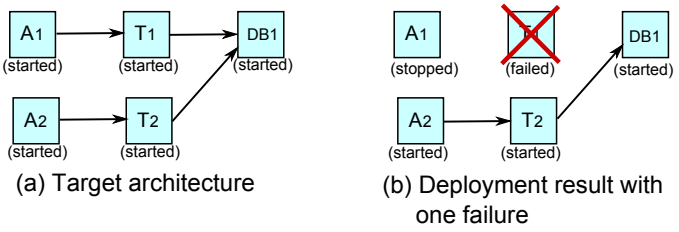

Fig. 6. Typical clustered Web architecture

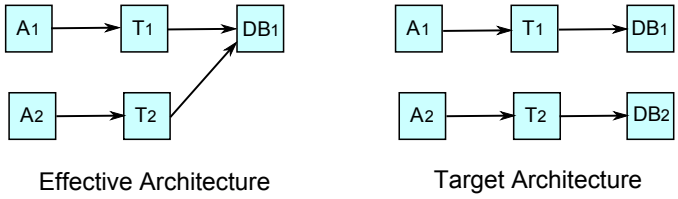

Fig. 7. Target architecture with two databases

represents a database server. All imports in the architecture are mandatory.

1) Deployment Scenario: The first management task is the initial deployment of the Web server. The administrator starts with an initial current architecture that is empty and shapes the desired target architecture depicted in Figure 6(a). When the administrator commits the session, the reconfiguration will result in the deployment of the overall system. The down phase of our protocol computes an empty Apply Down Set since the current architecture of the managed system is empty. The up phase of our protocol (see Listing 1) computes the following Apply Up Set:

Apply Up Set / commit algorithm

construct: T_1, T_2, DB_1
wire: (T_1, DB_1), (T_2, DB_1)
start: T_1, T_2, DB_1
construct: A_1, A_2
wire: (A_1, T_1), (A_2, T_2)
start: A_1, A_2

Our protocol then orders this Apply Up Set into the following ICS:

ICS / commit algorithm

construct: A_1, A_2, T_1, T_2, DB_1
wire: (A_1, T_1), (A_2, T_2), (T_1, DB_1), (T_2, DB_1)
start: DB_1, T_1, T_2, A_1, A_2

Without failures, the apply of this ICS results in the deployment of the desired clustered Web server. Let's now force a failure to occur upon wiring Tomcat to the database system (e.g., wire(T_1, DB_1)). The administrator may choose to suspend the reconfiguration session, knowing that the resulting system is consistent and reconfigurable. In this particular instance, the administrator would probably prefer to use the Roll-Forward Policy (RFP_commit algorithm in Listing 3) because isolated failures during a deployment usually do not justify to rollback the entire deployment.

The Roll-Forward policy executes the recover_current algorithm (Listing 2) that accounts for the failure of the Tomcat instance T_1 and produces the following ICS: fail(T_1). Indeed, propagating the failure of T_1 has no effects since no components were wired to T_1 at the time it failed.

Then the Roll-Forward policy executes the recover_target algorithm (Listing 3) that accounts for the failure of T_1 in the target architecture. T_1 is marked as failed, which propagates in isolating T_1 ($failed$ propagation rule). Removing mandatory wires to T_1 propagates into stopping components depending on these wires ($unwired$ propagation rule). Ultimately, this produces the following ICS: stop(A_1), unwire(A_1, T_1), fail(T_1), leading to the target architecture that is depicted in Figure 6(b).

Committing this recovered target architecture allows to get a managed system that is running despite a partial failure during its deployment. Moreover, it is consistent and thus ready to be reconfigured again.

2) Sizing Scenario: We consider adding a new database server (DB_2) and balancing the Tomcat servers over the two database servers, as depicted in Figure 7. Being given the current and target architectures of Figure 7, our protocol computes the following Apply Down Set and Apply Up Set:

Apply Down Set	*Apply Up Set*
- *unwire:* (T_2, DB_1)	- *construct:* DB_2
	- *start:* DB_2
	- *wire:* (T_2, DB_2)

Then, through propagation and ordering, our protocol generates the ICS given below which is a longer ICS because the *unwire*(T_2, DB_1) propagates a *stop* operation on both the Tomcat and Apache components. They are also longer because of the larger up set to reach the target architecture. Indeed, since the protocol just stopped a Tomcat and an Apache that are not stopped in the target architecture, the diff for the up phase will produce the extra *start* operations needed.

Down ICS	*Up ICS*
- *stop:* A_2, T_2	- *construct:* DB_2
- *unwire:* (T_2, DB_1)	- *wire:* (T_2, DB_2)
	- *start:* DB_2, T_2, A_2

Without failure, the reconfiguration achieves sizing up the clustered Web server. Let's now force a first failure that occurs on the start on DB_2. The recover_current algorithm computes the following ICS: unwire(T_2, DB_2), fail(DB_2).

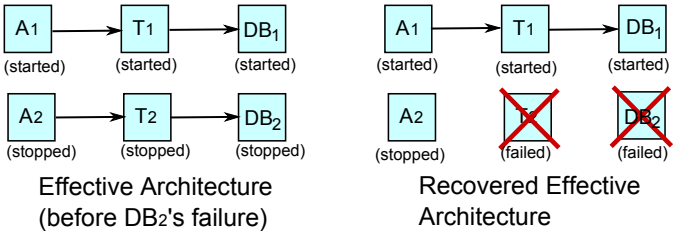

Fig. 8. DB2's failure during reconfiguration

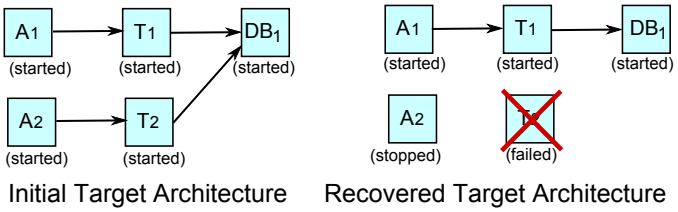

Fig. 9. Roll-Backward policy on DB2's failure

This ICS would recover a consistent architecture if applied entirely on the managed system. However, we will interrupt it, considering the case of a cascading failure when attempting to *unwire* the Tomcat instance T_2. The fixpoint on the recover_current generates a new ICS: unwire(A_2, T_2), fail(T_2), fail(DB_2). This time, we will consider that the apply of this ICS succeeds, meaning that the *unwire* on the Apache instance A_2 succeeds and the recovery completes with a final fail set of $\{T_2, DB_2\}$.

The Figure 8 shows the architecture of the managed system before the first failure (occurring on the start of DB_2) and after the recovery process. Notice the post-recovery architecture is architecturally consistent; the administrator can therefore introspect the architecture, discover and understand the failures, fix the reasons for the failures, if any, and ultimately issue one or more reconfigurations to reach an acceptable configuration of the overall system.

D. Protocol Scalability

In this section, we wish to discuss the scalability of our protocol in terms of the number of reconfiguration operations it issues.

Without considering failures, our protocol issues a number of reconfiguration operations that is solely governed by the complexity of the current and target architectures and their relative distance. Given a current architecture A_C and a target architecture A_T, the largest possible reconfiguration is to destroy entirely A_C and thereby entirely construct A_T. In this worst case scenario, our protocol issues a maximum number of operations that is less or equal than $N_{destruct}$+$N_{construct}$ where $N_{destruct}$ is the number of operations needed to entirely destruct A_C and $N_{construct}$ is the number of operations needed to entirely construct A_T.

More precisely, given an architecture with C component and W wires, the $N_{destruct}$ and $N_{construct}$ numbers must be smaller that $2C + W$. Indeed, regarding $N_{destruct}$, each component may at most be STOPPED and DESTRUCTED, while each wire may at most be UNWIRED. Regarding $N_{construct}$, each component may at most be CONSTRUCTED and STARTED, while each wire may at most be WIRED.

Notice that the saturation of the Apply Down Set, which is a fixpoint, does not change the evaluation of this maximum number of operations. Indeed, at the most, this saturation may only add one unwire operation per wire and one stop operation per component, operations that are already included in $N_{destruct}$.

When considering failures, our protocol goes through a recovery fixpoint that propagates the impacts of failures throughout the current architecture. In the worst case, all components will incrementally fail, in which case, the recovery propagation will issue at most one unwire operation per wire and one stop and fail operations per component. Consequently, given an architecture A with C component and W wires, we can define $N_{recover}(A)$ as equal to $2C + W$. However, since failures may occur at any time, we need to consider the number $N_{recover}$ that is equal to the maximum of the $N_{recover}(A)$ for all architectures the managed system is going through from A_C to A_T.

Hence, even when considering failures, our protocol issues less operations than $N_{destruct} + N_{construct} + N_{recover}$, a number of operations that is still linear with the complexity of A_C and A_T architectures and their relative distance. In the use-case given in Section VI-C1, $N_{destruct}$ is null as the initial system is empty and $N_{construct}$ is equal to 14 (5 CONSTRUCT, 4 WIRE, and 5 START) and $N_{propagate}$ is also equal to 14 (4 UNWIRE, 5 STOP, and 5 FAIL).

VII. Related Work

Many component models define a component lifecycle with two states (started and stopped) controlled by the mandatory and optional semantics of wires [8][9][10][25]. They provide reconfiguration operations that add/remove components and wires between components. However, these operations are invoked individually, with an immediate effect on the component assembly. This means that anyone reconfiguring a component assembly must take care of the following complex and error-prone tasks. First, (s)he must apply reconfiguration operations in the correct order. For instance, only start a component after it has all its mandatory imports wired). Second, (s)he must manually propagate the collateral effects of each reconfiguration operation, such as stopping a client component before stopping a server one). With a model-driven reconfiguration approach, one simply shapes the target architecture, leaving these tasks to the reconfiguration protocol.

Regarding approches that advocate model-driven reconfigurations [14][13][15][19][21][27], the focus is mainly on on the capture of application-specific architectural constraints, mostly through the use of architectural styles [16][26]. Complementary to the architectural constraints that we consider

in this paper, application-specific constraints do not require to be incrementally preserved during a reconfiguration, they only need to be preserved by the target architecture. In other words, application-specific constraints intend to shape *functional* architectures while architectural constraints guarantee *reconfigurable* architectures.

Most of the above frameworks did not publish any details on their reconfiguration protocol, with the noticeable exception of [21]. Their protocol orders reconfiguration operations following [17], only considering optional wires between components—a choice that is consistent with their underlying component platform [22]. Regarding failures, only a few frameworks [13][14][19] published about fault-tolerance, all advocating a rollback strategy based on the use of inverse reconfiguration operations; a design that only supports a single failure per reconfiguration.

[2] discusses failures occurring during recovery and states that it is a hard challenge for autonomic systems. However, the paper only presents early ideas, suggesting that component dependencies can be used to plan the propagation of the impacts of failures throughout a component assembly. The paper also suggests that recovery should be structured as a fixpoint, but does not give any details. In particular, no algorithms are described.

The RAPIDware project [28] is another component framework that shares similarities with our proposal. It proposes a formal model [18] to verify that during and after reconfigurations, the system remains in correct states in terms of architectural and behavioral invariants. As [19], it proposes a reconfiguration process that handles failures that appear at commit time through inverse operations.

Finally, in a prior work [7], we related our verification experience of an earlier version of the reconfiguration protocol. The focus of the paper was the formal verification of our protocol using the CADP toolbox [12], only briefly introducing our protocol as an example of a verification process of interest to the community of formal methods. The paper only sketched our protocol, including the notion of ICS and its associated ordering algorithm (depicted in Fig 4). In contrast, the presented paper discusses the complete protocol at length, including its detailed design and detailed algorithms. In particular, we acknowledge for the first time the novel ordering of operations based on four steps: architectural diff, saturate, order, and apply. Moreover, this paper also reports on proving the protocol rather than verifying it, providing on-line the complete specification in Coq. Finally, this paper includes an evaluation of the protocol.

VIII. Conclusion

This paper summarized the growing consensus of modern component models and the corresponding reconfiguration contract: reconfiguration grammar and architectural invariants. It proposed a reconfiguration protocol based on the concept of Incrementally Consistent Sequences (ICS), ensuring that any reconfiguration incrementally respects this reconfiguration contract. The proposed protocol resists any number of failures during the reconfiguration, always producing an architecturally consistent assembly of components that can be safely introspected and further reconfigured. In that regards, we proposed two advanced recovery policies, the Roll-Backward policy that rolls back a failed reconfiguration and the Roll-Forward policy that pushes towards the desired target architecture, both policies going as far as possible, failure permitting. We fully specified our protocol and proved it correct using the Coq proof assistant and we evaluated its complexity (linear with respect to the complexity of the desired reconfiguration).

Acknowledgements

The work described in this paper was partially supported by FSN project OpenCloudWare.

References

[1] Coq formalisation and certification of the presented reconfiguration algorithms,http://sardes.inrialpes.fr/ pous/rrca/.

[2] N. Arshad, D. Heimbigner, and A. L. Wolf. Dealing with failures during failure recovery of distributed systems. *SIGSOFT Softw. Eng. Notes*, 30:1–6, May 2005.

[3] N. Bencomo, P. Grace, C. A. Flores-Cortés, D. Hughes, and G. S. Blair. Genie: supporting the model driven development of reflective, component-based adaptive systems. In *Proc. ICSE'08*, 2008.

[4] Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. Springer, 2004.

[5] G. S. Blair, N. Bencomo, and R. B. France. Models@run.time. *IEEE Computer*, 42(10):22–27, 2009.

[6] Sara Bouchenak, Fabienne Boyer, Benoit Claudel, Noel De Palma, Olivier Gruber, and Sylvain Sicard. From autonomic to self-self behaviors: The jade experience. *TAAS*, 6(4):28, 2011.

[7] F. Boyer, O. Gruber, and G. Salaun. Specifying and Verifying a Robust Reconfiguration Protocol with LOTOS NT/CADP. In *17th Int. Symp. on Formal Methods (FM'11)*, 2011.

[8] E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, and J.B. Stefani. The Fractal Component Model and its Support in Java. *Software – Practice and Experience (SP&E)*, 36(11-12):1257–1284, September 2006.

[9] G. Coulson, G.S. Blair, M. Clarke, and N. Parlavantzas. The design of a configurable and reconfigurable middleware platform. *Distributed Computing*, 15(2):109–126, 2002.

[10] C. Escoffier, R. S. Hall, and P. Lalanda. ipojo: an extensible service-oriented component framework. In *IEEE Int. Conf. on Services Computing (SCC 2007)*, 2007.

[11] J. Floch, S. Hallsteinsen, E. Stav, F. Eliassen, K. Lund, and E. Gjorven. Using architecture models for runtime adaptability. *Software, IEEE*, 23(2):62 – 70, march-april 2006.

[12] H. Garavel, R. Mateescu, F. Lang, and W. Serwe. CADP 2006: A Toolbox for the Construction and Analysis of Distributed Processes. In *Proc. CAV'07*, pages 158–162. Springer, 2007.

[13] J. C. Georgas, A. van der Hoek, and R. N. Taylor. Architectural runtime configuration management in support of dependable self-adaptive software. In *Workshop on Architecting Dependable Systems*, WADS '05, 2005.

[14] A. Gomes, Tadeu A., T. Batista, A. Joolia, and G. Coulson. Architecting dependable systems iv. chapter Architecting dynamic reconfiguration in dependable systems, pages 237–261. Springer-Verlag, Berlin, Heidelberg, 2007.

[15] A. Joolia, T. Batista, G. Coulson, A. Tadeu, and A. Gomes. A.t.a.: Mapping adl specifications to an efficient and reconfigurable runtime component platform. In *IEEE/IFIP Conference on Software Architecture (WICSA'05)*, 2005.

[16] J. S. Kim and D. Garlan. Analyzing architectural styles. *J. of Systems and Software*, 83(7):1216–1235, 2010.

[17] J. Kramer and J. Magee. The evolving philosophers problem: Dynamic change management. *IEEE Trans. Software Eng.*, 16(11):1293–1306, 1990.

[18] S. S. Kulkarni and K. N. Biyani. Correctness of component-based adaptation. In *7th Int. Symp. on Component-Based Software Engineering (CBSE'04)*, 2004.

[19] M. Léger, T. Ledoux, and T. Coupaye. Reliable dynamic reconfigurations in a reflective component model. In *13th Int. Symp. on Component-Based Software Engineering (CBSE'10)*.

[20] B. Morin, O. Barais, J-M Jézéquel, F Fleurey, and A Solberg. Models at Runtime to Support Dynamic Adaptation. *IEEE COMPUTER*, pages 46–53, 2009.

[21] B. Morin, O. Barais, G. Nain, and J.-M. Jezequel. Taming dynamically adaptive systems using models and aspects. In *Proc. ICSE'09*, 2009.

[22] OSGi. OSGi service platform, release 4.1, OSGi Comittee, http://www.osgi.org, 2008.

[23] R. Roshandel, A. Van Der Hoek, M. Mikic-Rakic, and N. Medvidovic. Mae – a system model and environment for managing architectural evolution. *ACM TRANS. SOFTW. ENG. METH.*, 13(2):240–276, 2004.

[24] S. Sicard, F. Boyer, and N. De Palma. Using components for architecture-based management: the self-repair case. In *Proc. ICSE'08*, 2008.

[25] Spring. The Spring Framework, http://www.springframework.org, 2006.

[26] R. N. Taylor, N. Medvidovic, and P. Oreizy. Architectural styles for runtime software adaptation. In *IEEE/IFIP Conf. on Software Architecture (WICSA'09)*, pages 171–180, 2009.

[27] G. Waignier, P. Sriplakich, A.-F. Le Meur, and L. Duchien. A model-based framework for statically and dynamically checking component interactions. In *Proc. MODELS'08*, volume 5301, pages 371–385. Springer, 2008.

[28] J. Zhang and B. H. C. Cheng. Model-based development of dynamically adaptive software. In *Proc. ICSE'06*, pages 371–380, New York, NY, USA, 2006. ACM.