ELSEVIER

Electronic Notes in Theoretical Computer Science 260 (2010) 91–108

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

Correct Execution of Reconfiguration for Stateful Components*

Moritz Hammer

Ludwig-Maximilians-Universität München
hammer@ifi.lmu.de

Alexander Knapp

Universität Augsburg
knapp@informatik.uni-augsburg.de

Abstract

In component-based software engineering, reconfiguration describes structural changes to the architecture of a component system. For stateful components, not only structural but also behavioural aspects have to be taken into account in reconfiguration. We present a procedure to conduct reconfiguration in systems of concurrent, stateful components that interferes as little as possible with unchanged subsystems. Reconfiguration is described by a plan for adding, deleting and reconnecting components. A plan is executed by a sequence of simple, local steps, which are suitable for implementation in a programming language. We prove that plan execution is indistinguishable from atomic reconfiguration and use this fact for state-space reduction for verifying properties by model checking.

Keywords: Reconfiguration, stateful components, model checking

1 Introduction

Software components have been proposed as a useful level of system partitioning ever since it became evident that such a partitioning would be necessary to understand and maintain larger software systems [11]. They are especially appealing if the configuration of components is done independently from the definition of the components themselves. It thus becomes possible to reason about the design of a software system on a level higher than actual code. It also becomes easier to reason about a redesign of the system at runtime, a process called reconfiguration [8].

We investigate a framework for reconfiguration of stateful components. To describe reconfiguration, we employ *plans*, which are put into effect by a sequence

* This work has been partially sponsored by the project SENSORIA IST-2005-016004.

1571-0661 © 2009 Elsevier B.V. Open access under CC BY-NC-ND license.
doi:10.1016/j.entcs.2009.12.033

of configuration rewriting steps. These steps are fine-grained enough to be implemented in a common programming language without having to block independent components.

Avoiding any unnecessary blocking is an important aspect of reconfiguration, since quite often, only a small part of the component system is modified, and blocking of the entire, possibly distributed system is infeasible. At the same time, executing reconfiguration in a highly concurrent scenario is hard, as it must not interfere with the normal component behaviour. Reconfiguration should hence be "perceived as atomic", which describes that any component not directly involved in the reconfiguration should either detect that it is operating in the old or in the new configuration, but not in any intermediate state. Combining both the "minimal invasiveness" and the "perceived atomicity" criteria are requirements for a *lock-free algorithm* [7].

In our case, reconfiguration is further complicated by the fact that the state of old components needs to be transferred to new components, which cannot be done automatically in every situation [21]. Like all other state-aware reconfiguration frameworks we know of, we pass this problem on to the user, who is required to explicitly describe how the state is to be transferred. This puts the user into an inconvenient situation: depending on the cause of the reconfiguration, state transferal can become arbitrarily complex (maybe a faulty component needs to be replaced, and its state has already become corrupted). At the same time, reconfiguration might occur only very infrequently, and thus be difficult to test.

We hence employ formal techniques to facilitate the task of defining a reconfiguration, and subject the plan rewriting rules to model checking. Here, the perceived atomicity becomes important since treating reconfiguration as a single step is much less costly than to have it distributed over a series of steps, with the part of the system uninfluenced by reconfiguration executing interleaving steps.

The remainder of this paper is structured as follows: Related work is discussed in Sect. 2. In Sect. 3 we illustrate our idea of reconfiguring stateful components by means of a fault-tolerance example. In Sect. 4 we give a formal account of our framework for stateful components. Reconfiguration is described in Sect. 5, introducing a notion of reconfiguration plan and how such a plan is executed by simple, local steps. In particular, we prove that the execution of a so-called shallow reconfiguration plan is not observable by components not mentioned in the plan. We apply this result to the verification of the fault-tolerant components example in Sect. 6. In Sect. 7 we close by drawing conclusions and an outlook to future work.

2 Related Work

Many component frameworks supporting reconfiguration have been developed, for overviews see [22,13,4]. Those that consider the component state and its retainment during reconfiguration are primarily concerned with the technical process and the algorithms; theoretical considerations, like [17,21], have only been concerned with particular aspects of stateful reconfiguration. We are not aware of frameworks with

a comprehensive formal basis that consider local component states.

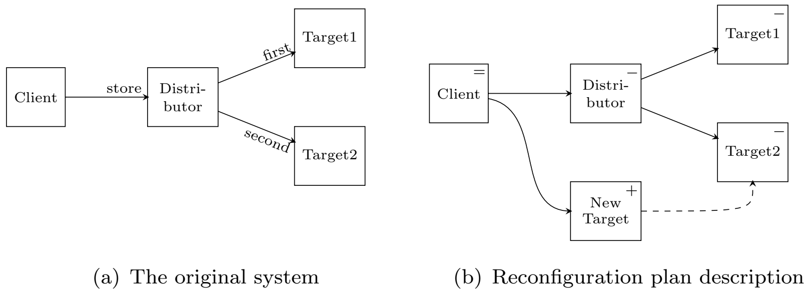
We give a brief overview of state-aware reconfiguration frameworks by their way of transferring the state. Argus [3] allows for the transferal of state with explicit queries to accessor functions. One of the most often cited works is Polylith [8], which requires components to implement encode and decode methods. The state can then be obtained from an old component and injected into a new one. Lim [12] uses similar state accessor methods, with a very broad definition of state. The work is otherwise similar to ours in that it proposes the employment of *schedules* to achieve successful reconfiguration. Bidan et al. [2] and Tewksbury et al. [20] both enhance Corba with the ability to do stateful dynamic updates (the practical example of [23] also uses Corba). The former requires state accessor methods in reconfigurable components, the latter work tries to transfer the state with a 1:1 mapping of variables, with a fallback to a user-defined mapping. Wegdam [23] employs a state translator, which is a function that is described as application dependent; this is very comparable to our approach. He uses a notion of "mutually consistent states", which provides an interesting criterion for verifying the correctness of state-considering reconfiguration, yet his work is not backed by a formal framework. In Casa [15], state transferal is triggered by the reconfiguration handler. They consider a very broad definition of components, however the notion of state remains quite abstract. Rasche and Polze [17] also employ a heuristic approach towards state transferal. Using an object-graph-traversal algorithm [18] and a mapping, they realize a semi-automatic, direct approach. For consistent state update, they rely on the well-known formally underpinned concept of *quiescence* [10]. Vandewoude [21] focuses on the transferal of state for component updates. His work is placed in the context of hot code updates, which gives some constraints on the data that need to be considered; a semi-automatic approach is presented that employs a series of strategies to copy the state. Quiescence is weakened to *tranquility* for obtaining safe reconfiguration states. CoBRA [9] uses the *memento pattern* to transfer the state, which amounts to encoding it into a special object and injecting it into the target component.

Most of these frameworks support interleaved reconfiguration, except [20], which prepares the reconfiguration concurrently, but performs the critical step of rewiring in an *atomic switch-over*. The other frameworks all use local blocking of the components that are to be removed, but a formally justified atomicity criterion is usually not given.

Model checking of component systems is done by the Vecors platform [1] and for the SOFA component framework [16], but although the latter supports reconfiguration, these two features have not been combined.

3 Example: A Stateful, Fault-Tolerant Component

We illustrate our approach to reconfiguration of stateful components by means of a small fault-tolerance example that nevertheless shows the capability of the method as well as the problems involved. Consider a component that stores data, maybe



(a) The original system (b) Reconfiguration plan description

Fig. 1. Fault tolerance components

an accounting system. We know that this component fails sometimes, and if it does, its state gets corrupted (i.e., we cannot make any safe assumption about it). We assume, however, that the component is deterministic, and that an error is communicated back to the callee immediately (e.g., in an actual implementation, by throwing an exception).

The basic idea (see Fig. 1(a)) is to maintain two copies of the unreliable component (Target1 and Target2), and to issue calls (from Client) to both of them (by Distributor); the second copy only receives the call after the call to the first copy has been completed successfully. Hence, any call issued to the second copy will successfully complete, so the state remains intact. Upon detecting a failure of the first copy, reconfiguration is employed to substitute the storage subsystem by another component (NewTarget, see Fig. 1(b)), which might be a different, maybe less efficient implementation, or some stub that takes care of a graceful and recoverable system shutdown.

Reconfiguration is done in several, local steps: First, the NewTarget component is instantiated, and the client's connection to Distributor is reconnected to it. Thus, any message sent by Client during the remainder of the reconfiguration will be stored in component NewTarget. By initialising NewTarget in an inactive state, messages are not yet consumed. The reconfiguration then proceeds to update NewTarget with the data of Target2 and possibly pending messages of Distributor. Thus, the state is retained, and no messages are lost. Finally, NewTarget is activated and the now disconnected components are terminated.

The pseudo-code for the example is given in Fig. 2. $\mathscr{S}$ and $\mathscr{V}$ are the sets of component states and parameters. It is important to fix the set $E_m \subseteq \mathscr{V}$ such that the Unsafe components (Target1, Target2) are fully deterministic: the second copy must not fail if the first one succeeded for a given parameter.

4 Component Framework

We describe a formal framework for concurrent, message-passing components with local state; it is inspired by the many already existing component frameworks (see [11] for an overview). The framework tries to restrict itself to just the elements required to illustrate our idea of reconfiguration of stateful components. In particular, component behaviour is described by a simple process language which

focuses on local state manipulation and message passing. These processes could also be expressed in other process languages like the π-calculus [14], but the issues of state and message queues would have to be encoded.

We assume a set $\mathscr{C}$ of *component identifiers*, a set $\mathscr{M}$ of *method names*, that are implemented by components, a set $\mathscr{R}$ of *rôle names*, through which components access other components, and a set $\mathscr{V}$ of *values* which are communicated between components. *Interfaces*, comprised in a set $\mathscr{I}$, are finite subsets of method names.

The *local component states* are drawn from a set $\mathscr{S}$ together with state manipulating functions: $upd : \mathscr{S} \times \mathscr{S} \to \mathscr{S}$ for updating a state with another state; $prm : \mathscr{S} \times \mathscr{M} \times \mathscr{V} \to \mathscr{S}$ for storing a parameter value for a method name; and $ret : \mathscr{S} \times \mathscr{V} \to \mathscr{S}$ for storing a returned value.

Definition 4.1 (Component process term) *The* implementation *of a method (name) in a component is described by a* component process term *in the set $\mathscr{P}$ defined by the following grammar:*

$$\begin{aligned} P \in \mathscr{P} ::= \; & \mathrm{call}(r, m, v).P \mid \mathrm{send}(r, m, v).P \mid \mathrm{return}(v).P \\ & \mid \mathrm{set}(\sigma).P \mid \mathrm{choose}((\Sigma_j.P_j)_{j \in J}) \mid \mathrm{success} \mid \mathrm{fail} \end{aligned}$$

with $r \in \mathscr{R}$, $m \in \mathscr{M}$, $v \in \mathscr{V}$, $\sigma \in \mathscr{S}$, $\Sigma_j \subseteq \mathscr{S}$, and J an index set.

In a component process term, $\mathrm{call}(r, m, v)$ represents a synchronous invocation of method m on the rôle r with parameter value v, where synchronous means that the

```
method Distributor::sync(prm)
  ret ← call(first, sync, prm)
  if (ret = err) then fail
  else call(second, sync, prm)
       return ret; success fi

method Distributor::async(prm)
  ret ← call(first, async, prm)
  if (ret = err) then fail
  else call(second, async, prm)
       success fi

method Unsafe::sync(prm)
  if (prm ∈ E_sync)
  then state ← err; return err
  else state ← su^sync(state, prm)
       return sr(state, prm) fi
  success

method Unsafe::async(prm)
  if (prm ∈ E_async)
  then state ← err; return err
  else state ← su^async(state, prm)
       return * fi
  success

method Safe::sync(prm)
  state ← su^sync(state, prm)
  return sr(state, prm); success

method Safe::async(prm)
  state ← su^async(state, prm)
  success
```

where, for $m \in \{\mathrm{sync}, \mathrm{async}\}$, $E_m \subseteq \mathscr{V}$ are sets of error-inducing parameter values and $su^m : \mathscr{S} \times \mathscr{V} \to \mathscr{S}$ are functions that produce the updated state of a component as the effect of method m, and $sr : \mathscr{S} \times \mathscr{V} \to \mathscr{V}$ is a function yielding the result of sync.

Fig. 2. Behaviour of the fault tolerance example

caller becomes blocked and waits for an answer by a $\mathrm{return}(v')$ with value v' from the callee. In contrast, $\mathrm{send}(r, m, v)$ asynchronously sends method name m with parameter value v to rôle name r and does not wait for a result. The action $\mathrm{set}(\sigma)$ updates the local component state with σ, the action $\mathrm{choose}((\Sigma_j.P_j)_{j\in J})$ selects non-deterministically some $j \in J$ such that the current local component state is in Σ_j and proceeds with P_j. Finally, success and fail represent the successful or abnormal termination of executing a method.

Example 4.2 The pseudo-code used in Fig. 2 can be translated to component process terms in a straight-forward manner: We define the set of data values $\mathscr{V}$ to include $*$ as a "don't care" and *err* as an error element. We define the set of local component states $\mathscr{S}$ as $S \times \mathscr{V} \times \mathscr{V}$ with S a set, again with $\{*, \mathit{err}\} \subseteq S$, and we let state, prm and ret mean the first, second, and third part of a local component state. Accordingly, the state manipulating functions are: $\mathit{upd}((s, v_{\mathrm{prm}}, v_{\mathrm{ret}}), (s', v'_{\mathrm{prm}}, v'_{\mathrm{ret}})) = (s', v_{\mathrm{prm}}, v_{\mathrm{ret}})$, $\mathit{prm}((s, v_{\mathrm{prm}}, v), m, v'_{\mathrm{prm}}) = (s, v'_{\mathrm{prm}}, v)$ and $\mathit{ret}((s, v_{\mathrm{prm}}, v), v') = (s, v_{\mathrm{prm}}, v')$. Then, we can translate, e.g., return ret resp. call(r, m, prm) to

$$\mathrm{choose}((\{(s, v_{\mathrm{prm}}, v_{\mathrm{ret}}) \mid s \in S, v_{\mathrm{prm}} \in \mathscr{V}\}.\mathrm{return}(v_{\mathrm{ret}}))_{v_{\mathrm{ret}}\in\mathscr{V}}) \text{ resp.}$$
$$\mathrm{choose}((\{(s, v_{\mathrm{prm}}, v_{\mathrm{ret}}) \mid s \in S, v_{\mathrm{ret}} \in \mathscr{V}\}.\mathrm{call}(r, m, v_{\mathrm{prm}}))_{v_{\mathrm{prm}}\in\mathscr{V}}) \ .$$

Definition 4.3 (Component) A *component* c is a tuple

$$(\mathit{id}(c), I_P(c), I_R(c), \mu(c), \iota(c))$$

with $\mathit{id}(c) \in \mathscr{C}$ the *component identifier*, $I_P(c) \subseteq \mathscr{I}$ a finite set of *provided interfaces*, $I_R(c) : \mathscr{R} \rightharpoonup \mathscr{I}$ a partial function with finite domain identifying *required interfaces* by rôle names, $\mu(c) : \bigcup I_P(c) \to \mathscr{P}$ a *method environment* assigning an implementation to each method the component provides, and $\iota(c) \in \mathscr{S}$ the *initial state*. The set of required rôle names $\mathrm{dom}(I_R(c))$ is denoted by $\mathscr{R}(c)$.

Definition 4.4 (Component configuration) A *component configuration* $\tilde{c}$ of a single component c is of the form

$$\mathit{id}(c), f(\tilde{c}), \gamma(\tilde{c}), e(\tilde{c}), \sigma(\tilde{c}), P(\tilde{c}), \pi(\tilde{c})$$

where $f(\tilde{c})$ indicates whether the component is *running* ($f(\tilde{c}) = \mathrm{r}$) or *blocked* ($f(\tilde{c}) = \mathrm{b}$); $\gamma(\tilde{c}) : \mathscr{R} \rightharpoonup \mathscr{C}$ is a wiring for the required interfaces of c, telling which rôle name points to which component; $e(\tilde{c})$ either contains a method call of the form $c'.r'.m(v)$ with $c' \in \mathscr{C}$, $r' \in \mathscr{R}$, $m \in \mathscr{M}$ and $v \in \mathscr{V}$, which c currently executes, or is empty ($\perp$); $\sigma(\tilde{c}) \in \mathscr{S}$ is a component state; $P(\tilde{c}) \in \mathscr{P}$ is a component process term; and $\pi(\tilde{c}) \in (\mathscr{C} \times \mathscr{R} \times \mathscr{M} \times \mathscr{V})^*$ represents the message queue, i.e., a sequence of messages, where we write :: for concatenation and ε for the empty sequence. For ease of reading, we write c^f for $\mathit{id}(c), f(\tilde{c})$, and we group $e(\tilde{c})$ and $\sigma(\tilde{c})$ to form $\langle e(\tilde{c}), \sigma(\tilde{c})\rangle$.

We say that $\gamma(\tilde{c}) : \mathscr{R} \rightharpoonup \mathscr{C}$ is *well-connected* if for all rôles r defined in $\gamma(\tilde{c})$ the types of the provided and required interfaces match, i.e., $I_R(c)(r) \in I_P(\gamma(\tilde{c})(r))$;

$$c_1^{\mathrm{r}}, \gamma_1, \mathrm{call}(r, m, v).P \parallel c_2, \pi_2 \rightarrow c_1^{\mathrm{b}}, \gamma_1, P \parallel c_2, \pi_2 :: c_1.r.m(v) \quad \text{if } c_2 = \gamma_1(r) \qquad (\textsc{call})$$

$$c_1^{\mathrm{r}}, \gamma_1, \mathrm{send}(r, m, v).P \parallel c_2, \pi_2 \rightarrow c_1^{\mathrm{r}}, \gamma_1, P \parallel c_2, \pi_2 :: c_1.r.m(v) \quad \text{if } c_2 = \gamma_1(r) \qquad (\textsc{send})$$

$$c_1^{\mathrm{r}}, \langle c_2.r.m(v')\rangle, \mathrm{return}(v).P \parallel c_2^{\mathrm{b}}, \langle\sigma\rangle \rightarrow c_1^{\mathrm{r}}, \langle\bot\rangle, P \parallel c_2^{\mathrm{r}}, \langle ret(\sigma, v)\rangle \qquad (\textsc{return})$$

$$c^{\mathrm{r}}, \langle\sigma\rangle, \mathrm{set}(\sigma').P \rightarrow c^{\mathrm{r}}, \langle upd(\sigma, \sigma')\rangle, P \qquad (\textsc{set})$$

$$c^{\mathrm{r}}, \langle\sigma\rangle, \mathrm{choose}((\Sigma_j.P_j)_{j\in J}) \rightarrow c^{\mathrm{r}}, \langle\sigma\rangle, P_j \quad \text{if } j \in J \text{ and } \sigma \in \Sigma_j \qquad (\textsc{choose})$$

$$c^{\mathrm{r}}, \langle\sigma\rangle, \mathrm{success}, c'.r.m(v) :: \pi \rightarrow c^{\mathrm{r}}, \langle c'.r.m(v), prm(\sigma, m, v)\rangle, \mu(c)(m), \pi \qquad (\textsc{deq})$$

Table 1
Configuration transition rules

and we say that γ is *completely connected* if $\mathrm{dom}(\gamma(\tilde{c})) = \mathscr{R}(c)$.

Example 4.5 For the fault-tolerance example, let the interface I provided by the unreliable component consist of the methods sync for synchronous and async for asynchronous invocations; and let the four components involved in the initial component graph of Fig. 1(a) be given by

- $(\mathrm{Client}, \{\}, \{\mathrm{store} \mapsto \mathrm{I}\}, \mu_{\mathrm{Client}}, \iota_{\mathrm{Client}})$,
- $(\mathrm{Distributor}, \{\mathrm{I}\}, \{\mathrm{first} \mapsto \mathrm{I}, \mathrm{second} \mapsto \mathrm{I}\}, \mu_{\mathrm{Distributor}}, \iota_{\mathrm{Distributor}})$,
- $(\mathrm{Target1}, \{\mathrm{I}\}, \{\}, \mu_{\mathrm{Unsafe}}, \iota_{\mathrm{Target}})$ and $(\mathrm{Target2}, \{\mathrm{I}\}, \{\}, \mu_{\mathrm{Unsafe}}, \iota_{\mathrm{Target}})$.

with arbitrary initial states ι_{Client}, $\iota_{\mathrm{Distributor}}$, and ι_{Target}. For the method environment μ_{Client} we only assume that it keeps sending an arbitrary stream of sync and async messages to Distributor. The method environments $\mu_{\mathrm{Distributor}}$ and μ_{Unsafe} are specified in the upper part of Fig. 2.

Definition 4.6 (Configuration) A *configuration* $\tilde{C}$ of a finite set of components $C = \{c_1, \ldots, c_n\}$ is a map $\{c_1 \mapsto \tilde{c}_1, \ldots, c_n \mapsto \tilde{c}_n\}$ from components to component configurations; we write $\tilde{c}_1 \parallel \cdots \parallel \tilde{c}_n$ for such a map.

How configurations may change into other configurations is described by *configuration transition rules* which can be applied to a configuration $\tilde{C}$ to yield another configuration $\tilde{C}'$. When defining such a rule, we abbreviate all occurring configurations to focus on relevant parts, in particular those that are changed. For example, the rule

$$c^{\mathrm{r}}, \langle\sigma\rangle, \mathrm{set}(\sigma').P \rightarrow c^{\mathrm{r}}, \langle upd(\sigma, \sigma')\rangle, P$$

when applied to a configuration $\tilde{c}_1 \parallel \cdots \parallel \tilde{c}_n$ with $c_1 = c$ spells out to

$$c^{\mathrm{r}}, \gamma, \langle e, \sigma\rangle, \mathrm{set}(\sigma').P, \pi \parallel \tilde{c}_2 \parallel \cdots \parallel \tilde{c}_n \rightarrow c^{\mathrm{r}}, \gamma, \langle e, upd(\sigma, \sigma')\rangle, P, \pi \parallel \tilde{c}_2 \parallel \cdots \parallel \tilde{c}_n\ .$$

The configuration transition rules of our component framework are listed in Tab. 1

and reflect the informal semantics of the process actions by homonymous rules. In particular, CALL and SEND treat synchronous and asynchronous invocations differently by blocking resp. continuing the caller. Although, in principle, we do not distinguish asynchronous and synchronous methods in interfaces, we assume that a method knows how it is going to be called: if it is invoked by call, it eventually needs to process a return subterm in order to unblock the caller, and if it is invoked by send, it should not contain return subterms. By DEQ a component can start processing of a new message from its queue. Note that DEQ applies only to running components and therefore prohibits circular synchronous calls: if processing a synchronous call from component c to component c' requires a synchronous call from c' to c, the system deadlocks. We deliberately accept this, as it preserves an important invariant: during processing a method, the state is not externally modified.

5 Reconfiguration

Reconfiguration is the process of changing the component graph [24]. This involves the removal and addition of components (i.e., nodes of the graph) and rewiring the components (i.e., adding and removing edges of the graph). We will, however, consider a less generic approach only: During reconfiguration, a set of components is removed and a set of components is added; only the connections of a retained component to a removed component are rewired to a new component. More complex scenarios (like a retained component becoming rewired to another retained one) can be reduced to such a simple scenario by removing and re-adding one of the retained components — for a practical implementation, it is straightforward to identify and treat such “delete and add again” scenarios. As mentioned before, the challenge of reconfiguration of stateful components is given by the need to retain the state, which consists of the data state $\sigma \in \mathscr{S}$ and the message queue π in our concrete component framework.

We will first introduce reconfiguration plans that describe the effect of a reconfiguration. We then introduce a set of fine-grained rules and a plan implementation that describes how these rules are applied to conduct a reconfiguration. We proceed to show that, for plans that redistribute messages in canonical way, plan implementation is perceived as atomic, while stopping only a minimal number of components.

5.1 Reconfiguration Plans

A reconfiguration of a configuration $\tilde{C}$ with $C = \mathrm{dom}(\tilde{C})$ is described by a *plan*

$$\Delta = (A, R, \alpha, \rho, \delta, \varsigma)$$

with A being a set of components to be added where $A \cap C = \varnothing$; $R \subseteq C$ the set of components to be removed; and α, ρ, δ and ς functions that describe how components are connected (α for components in A, ρ for components in $C \setminus R$) and how the state is preserved (δ handling the messages and ς the data state).

Hence, $\alpha : A \to (\mathscr{R} \to (C \setminus R) \cup A)$ describes the connections of the new

components in A, which may be connected to both other new components and components that already exist, but do not get removed. We require $\alpha(c)$ to be well-connected for all $c \in A$. The partial function $\rho : C \setminus R \times \mathscr{R} \rightharpoonup A$ describes the rewiring of connections that get discarded because the target component is removed. Thus, ρ needs to be defined for $(c, r) \in C \setminus R \times \mathscr{R}$ iff $\gamma(\tilde{c})(r) \in R$. We require that $\gamma(\tilde{c})[r \mapsto \rho(c, r)]$, i.e., the connections of c with r pointing to $\rho(c, r)$, is well-connected for c for all r for which $\rho(c, r)$ is defined. Of course, if $\gamma(\tilde{c})$ is completely connected, then so is $\gamma(\tilde{c})[r \mapsto \rho(c, r)]$.

δ and ς are two functions that are used for defining how the state of an old component should be preserved. The partial function $\delta : R \times (C \times \mathscr{R}) \rightharpoonup A$ describes message retainment, i.e., the components that should process previously unprocessed messages of components in R. It is required that messages are moved to components that actually implement the required interface, that is $I_R(c)(r) \in I_P(\delta(c', (c, r)))$ for all $(c', (c, r)) \in \mathrm{dom}(\delta)$. The function $\varsigma : (R \to \mathscr{S}) \times A \to \mathscr{S}$ describes the state of the new components, which is calculated from the state of the old components. This is a very general notion which subsumes more concrete, technical approaches of data state retainment [21,18,9].

Example 5.1 In the fault-tolerance-example, when the component Distributor has moved to fail, a reconfiguration has to be launched. The plan to be employed is illustrated in Fig. 1(b) using a single push-out graph transformation rule as used in [19]; the dashed line is an "update edge" indicating state retrieval. In our notation, this plan $\Delta = (A, R, \alpha, \rho, \delta, \varsigma)$ is represented by

$$
\begin{aligned}
&A = \{(\mathrm{NewTarget}, \{\mathrm{I}\}, \{\}, \mu_{\mathrm{Safe}}, \iota_{\mathrm{Target}})\} \;, \\
&R = \{\mathrm{Distributor}, \mathrm{Target1}, \mathrm{Target2}\} \;, \quad \alpha = \{\} \;, \\
&\rho = \{\mathrm{Client}, \mathrm{store} \mapsto \mathrm{NewTarget}\} \;, \quad \varsigma = \{(r, \mathrm{NewTarget}) \mapsto r(\mathrm{Target2})\} \;. \\
&\delta = \{(\mathrm{Distributor}, (\mathrm{Client}, \mathrm{store})) \mapsto \mathrm{NewTarget}\}
\end{aligned}
$$

where the method environment μ_{Safe} is specified in the lower part of Fig. 2.

Reconfiguration may only commence if all the components of R are not executing a method, i.e., their process terms are either success or fail. This is similar to the *quiescent states* of [10], which, roughly speaking, represent a situation where no communication is interrupted by a reconfiguration.

When a plan Δ is applied to a configuration $\tilde{C}$, the component set C is partitioned in three sets: the set R of components that get removed, a set $W = \mathrm{dom}(\rho)$ of components that need to have a rôle rewired (by definition, W is disjoint to R), and the set $C \setminus (R \cup \mathrm{dom}(\rho))$ of components that are not modified at all; after reconfiguration, the set A is added. Hence we can describe the effect of reconfiguration by the rule

$$
\tilde{R} \parallel \tilde{W} \to \Delta(\tilde{W}) \parallel \tilde{A}_{\Delta} \qquad \text{(RECONF)}
$$

with all components in R not performing a method, i.e., $\tilde{R} = c_1^{\mathrm{r}}, P_1 \parallel \ldots \parallel c_n^{\mathrm{r}}, P_n$ with $P_i \in \{\mathrm{success}, \mathrm{fail}\}$ and $\Delta(\tilde{W})$ describing the effect of the rewiring as defined

by ρ and $\tilde{A}_\Delta$ consisting of new components, which are initialised using α, δ and ς. In particular, if $\tilde{W} = c_{n+1}, \gamma_{n+1} \parallel \ldots \parallel c_{n+m}, \gamma_{n+m}$, then $\Delta(\tilde{W}) = c_{n+1}, \gamma'_{n+1} \parallel \ldots \parallel c_{n+m}, \gamma'_{n+m}$ with

$$\gamma'_{n+i}(r) = \begin{cases} \rho(c_{n+i}, r), & \text{if } \rho(c_{n+i}, r) \text{ is defined} \\ \gamma_{n+i}(r), & \text{otherwise.} \end{cases}$$

The configuration of new components $\tilde{A}_\Delta$ is

$$\tilde{A}_\Delta = \{c \mapsto c^{\mathrm{r}}, \alpha(c), \langle \bot, \varsigma(\sigma_R, c)\rangle, \text{success}, \pi_c \mid c \in A\} \ .$$

Herein, σ_R is the function capturing the states of components in R, i.e., for $c' \in R$ and thus $\tilde{c}'$ being part of $\tilde{R}$, we have $\sigma_R(c') = \sigma(\tilde{c}')$. The message queue π_c is a linearisation of the parallelisation (or shuffling) of message sequences copied by δ: Let $m_e = e(\tilde{c}_t)$ if $P(\tilde{c}_t) = $ fail and $m_e = \varepsilon$ otherwise; and let $\pi|_\varphi$ denote the sequence of messages of a queue π which are in φ. If $(c_t, (c_s, r_i)) \in \delta^{-1}(c)$, then $m_e :: \pi(\tilde{c}_t)|_\varphi$ is a subsequence of π_c, for $\varphi \equiv \{c_s.r_i.m(v) \mid m \in \mathscr{M} \wedge v \in \mathscr{V}\}$, and π_c consists exactly of these subsequences. Note that the order of the subsequences is unspecified, which makes plan application nondeterministic. m_e is the message that produced a fail, and needs to be processed by the substituting component again.

The RECONF rule performs the entire task of reconfiguration at once, hence providing atomicity. We will now refine the rules into a sequence of much finer rules (which can rightfully be assumed to be atomic) and proceed to show that these rules can be applied in a way that is indistinguishable from the effect of the RECONF rule.

5.2 Rules for Reconfiguration

In order to build fine-grained rules, we extend the set of component running states to $\{\mathrm{n}, \mathrm{i}, \mathrm{r}, \mathrm{b}, \mathrm{s}, \mathrm{c}\}$. The intended state machine for a component can be seen in Fig. 3. The new states have the following semantics:

- n: A newly initialised component that needs to be connected to other components.
- i: Once connected, a new component is put into this state, which is used for retrieving data and messages from old components, thus initialising the new component.

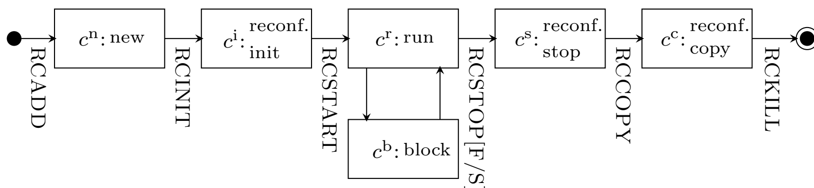


Fig. 3. State machine of a component, with rules to reach new states

$$\tilde{C} \to \tilde{C} \parallel c^{\mathrm{n}}, \gamma_\perp, \langle \iota(c) \rangle, \text{success}, \varepsilon \quad \text{if } \mathrm{dom}(\gamma_\perp) = \varnothing \tag{RCADD}$$

$$c_1^{\mathrm{n}}, \gamma_1 \parallel c_2 \to c_1^{\mathrm{n}}, \gamma_1[r \mapsto c_2] \parallel c_2 \quad \text{if } \gamma_1[r \mapsto c_2] \text{ is well-connected} \tag{RCWIRE}$$

$$c^{\mathrm{n}}, \gamma \to c^{\mathrm{i}}, \gamma \quad \text{if } \gamma \text{ is completely connected} \tag{RCINIT}$$

$$c^{\mathrm{i}} \to c^{\mathrm{r}} \tag{RCSTART}$$

$$c^{\mathrm{r}}, \text{success} \to c^{\mathrm{s}}, \text{success} \tag{RCSTOPS}$$

$$c^{\mathrm{r}}, \langle c_2.r.m(v) \rangle, \text{fail}, \pi \to c^{\mathrm{s}}, \text{fail}, c_2.r.m(v) :: \pi \tag{RCSTOPF}$$

$$c_1^{\mathrm{s}}, \gamma_1 \parallel c_2^{f_2}, \gamma_2 \parallel \ldots \parallel c_n^{f_n}, \gamma_n \to c_1^{\mathrm{c}}, \gamma_1 \parallel c_2^{f_2}, \gamma_2 \parallel \ldots \parallel c_n^{f_n}, \gamma_n \quad \text{if } \forall 2 \leq i \leq n \,.\, c_1 \in \mathrm{ran}(\gamma_i) \to f_i \in \{\mathrm{s}, \mathrm{c}\} \tag{RCCOPY}$$

$$\tilde{C} \parallel c^{\mathrm{c}} \to \tilde{C} \tag{RCKILL}$$

$$c_1, \gamma_1 \parallel c_2^{\mathrm{s}} \parallel c_3^{\mathrm{n}} \to c_1, \gamma_1[r \mapsto c_3] \parallel c_2^{\mathrm{s}} \parallel c_3^{\mathrm{n}} \quad \text{for } r \in \mathscr{R} \text{ with } \gamma_1(r) = c_2 \text{ if } \gamma_1[r \mapsto c_3] \text{ is well-connected} \tag{RCREWIRE}$$

$$c^{\mathrm{i}}, \langle \sigma \rangle \parallel c_1^{\mathrm{c}}, \langle \sigma_1 \rangle \parallel \ldots \parallel c_n^{\mathrm{c}}, \langle \sigma_n \rangle \to c^{\mathrm{i}}, \langle \varsigma((c_1 \mapsto \sigma_1, \ldots, c_n \mapsto \sigma_n), c) \rangle \parallel c_1^{\mathrm{c}}, \langle \sigma_1 \rangle \parallel \ldots \parallel c_n^{\mathrm{c}}, \langle \sigma_n \rangle \tag{RCSTATE}$$

$$c_1^{\mathrm{i}}, \pi_1 \parallel c_2^{\mathrm{c}}, \pi_2 \parallel c_3 \to c_1^{\mathrm{i}}, \pi_2|_\varphi :: \pi_1 \parallel c_2^{\mathrm{c}}, \pi_2|_{\neg\varphi} \parallel c_3 \quad \text{for } r \in \mathscr{R}(c_3) \text{ and } \varphi \equiv \{c_3.r.m(v) \mid m \in \mathscr{M}, v \in \mathscr{V}\} \tag{RCGETMSG}$$

Table 2
Reconfiguration transition rules

- s: Once a component is scheduled for removal, it is put into this state; it remains there until it has become entirely unconnected.
- c: Now that we are assured that no more messages are put into the queue from the outside (all components still connected to this component are in an s or c state), this state is taken, which allows the copying of parts of the message queue and the querying of the component state.

The fine-grained reconfiguration rules are shown in Tab. 2. There are two sets of rules: Rules that change the state of components to be added and to be removed, and rules that modify the components' connections and their data state. The former set consists of RCADD (adding a new component), RCINIT, RCSTART, RCSTOPS and RCSTOPF, RCCOPY, which can be applied to a stopped component once it is safe for having its state copied to other components, i.e., no active component's rôle points to the stopped component anymore and finally RCKILL, used to dispose of a component. (Note that RCCOPY needs to consider the entire configuration.)

The set of rules to change the state consists of RCWIRE to connect the rôles of a recently added component to other components; RCREWIRE to reconnect active components' rôles such that they point to new instead of stopped components; RCSTATE to take the states of the stopped components and combine them to a state for a new component; and RCGETMSG to transport residual messages of stopped to new components. RCSTATE is an abstraction of a sub-protocol that is performed to have the new component query the state of the old component; a description of

this process, which requires further component states and special restrictions on the process terms to avoid side-effects, exceeds the scope of this paper.

RCSTOPF is used to handle a component configuration with a fail process term; only this rule can advance such a configuration. Hence, fail is used to trigger a reconfiguration, which needs to follow a plan which disposes of the failed component. The method that failed must not be lost; maybe some other component is waiting for the return of the method, which would result in a deadlock. Thus, the method that produced the failure is prepended to the message queue, so that during reconfiguration, it can be moved to a new component which is capable of handling it properly.

5.3 Reconfiguration Plan Implementation

Given a plan $\Delta = (A, R, \alpha, \rho, \delta, \varsigma)$, we can *implement* Δ using a sequence of rule applications, which we call *actions*. We write $\text{RULE}(p_1 : z_1, \ldots, p_n : z_n)$ for an application of RULE, where the free variables $p_1, \ldots, p_n$ are instantiated with $z_1, \ldots, z_n$. We omit the p_i if the instantiation is clear from the context. For non-reconfiguration rules, only the acting component is free, except CHOOSE, where the chosen $j \in J$ is also free (since the choice might be nondeterministic).

(i) for each $c \in R$, $\text{RCSTOPS}(c)$ or $\text{RCSTOPF}(c)$ is used to stop the component. Note that this requires each $c \in R$ to eventually stop processing the current method. This may lead to deadlocks (if a component about to be reconfigured is blocked, but the target has already been stopped, with the synchronous method still in its queue); avoiding them is the responsibility of the plan deviser.

(ii) for each $c \in A$, we use $\text{RCADD}(c)$ to instantiate the component.

(iii) for each $c \in A$ and each $r \in \mathscr{R}(c)$, we use $\text{RCWIRE}(c, \alpha(c)(r))$, and for each $(c', r) \in \text{dom}(\rho)$, we use $\text{RCREWIRE}(c_1 : c', c_3 : \rho(c', r))$. We then use, for each $c \in A$, $\text{RCINIT}(c)$. This step connects the new and disconnects the old components.

(iv) then, we use $\text{RCCOPY}(c)$ for each $c \in R$. For each $(c', (c, r)) \in \text{dom}(\delta)$, we use $\text{RCGETMSG}(c_1 : \delta(c', (c, r)), c_2 : c', c_3 : c, r : r)$, thus copying all messages sent to c' over the rôle r from component c to a new component.

(v) for each $c \in A$, we use $\text{RCSTATE}(c, c_1, \ldots, c_n)$ for $\{c_1, \ldots, c_n\} = R$.

(vi) now, for each $c \in R$, $\text{RCKILL}(c)$ is used to remove the component, and, for each $c \in A$, $\text{RCSTART}(c)$ is used to start the components.

5.4 Shallow Reconfiguration Plans

The generic definition of δ is done to allow for arbitrary message retainment. A reconfiguration may require to remove a large set of components, some of which are only connected from components that also get removed. For those components (e.g., component O_2 in Fig. 4), a canonical message redistribution cannot be given; the unprocessed messages might have to be transferred to a new component that acts as a replacement. Such "deep" reconfiguration, however, is a complicated case that can

be expected to be of little interest. Most of the time, a single layer of components needs to retain messages, and the replacement can be found out from ρ. We thus define a *shallow reconfiguration plan* as $\Delta_s = (A, R, \alpha, \rho, \varsigma)$ which translates to a plan $\Delta = (A, R, \alpha, \rho, \delta^\rho, \varsigma)$ with

$$\delta^\rho(c', (c, r)) = \begin{cases} \rho(c, r), & \text{if } c \in C \setminus R \\ \text{undefined}, & \text{if } c \in R. \end{cases}$$

Hence, messages are moved "with the rewiring", which, due to its well-connectedness also ensures that the messages can indeed be processed. However, there is a problem involved if a new component gets connected to by more than one component; the order of the messages moved to the new component cannot be determined deterministically. Fig. 4 illustrates this problem: N becomes a replacement for both O_1 and O_3, so it receives the messages sent by C, but the order is arbitrary. In order to avoid this situation, for shallow plans we assume that $|\rho^{-1}(c)| \leq 1$ for all $c \in A$, i.e., a new component is only pointed to by one old component at most. This avoids the nondeterminism introduced by the arbitrary order in which RCGETMSG rules are executed.

Example 5.2 For the message retainment function δ and the rewiring function ρ of the plan of the fault-tolerance example $\delta = \delta^\rho$ holds and thus this plan can be represented by a shallow plan.

5.5 Interleaved Execution of Shallow Reconfiguration Plans

A *plan execution* for a plan Δ is a sequence of steps that are conducted following the rules of a plan implementation of Δ. An *interleaved* plan execution is a sequence of steps that are taken according to both normal and reconfiguration rules, with the steps conducted by reconfiguration rules following the plan implementation of Δ. Such an interleaved plan execution is a trace of a component system that gets reconfigured by Δ, while the components not in $A \cup R$ continue their execution.

Example 5.3 During the interleaved plan execution of the fault-tolerance example, the client may continue to issue asynchronous messages to the component connected

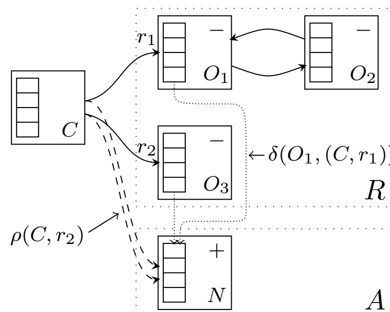


Fig. 4. A reconfiguration scenario illustrating the problems with message retainment: Preserving message order (O_1,O_3) and finding a destination (O_2). Non-solid arrows represent the functions ρ and δ of a plan $\Delta = (\{N\}, \{O_1, O_2, O_3\}, \{\}, \rho, \delta, \varsigma)$.

to its store rôle. While reconfiguring, not only the untainted state of Target2 is copied to NewTarget, but also those messages that have not yet been processed by Distributor are moved to NewTarget (due to using a shallow plan, they follow the rewiring), including those issued by Client in the time period between starting the reconfiguration and rewiring the store rôle. Furthermore, this also contains the message that triggered the error in Target1; the application of RCSTOPF re-enqueues this message such that it is not lost.

An important property of our approach is to ensure that the reconfiguration remains local; i.e., only a part of the component system is concerned. Thus, while a plan execution may be mixed with arbitrary steps of other components, these other components do not observe the reconfiguration until it is completely finished. As mentioned in the introduction, this "hot reconfiguration" requires some careful treatment of the components' states. For shallow plans, messages are transported in accordance to the rewiring of the retained components. Since this puts the messages sent before and after the application of RCGETMSG to the same component, we can show that reconfiguration of shallow plans is indeed observed as atomic.

In more detail, we prove that an interleaved plan execution of a shallow plan Δ_s can be simulated by another interleaved plan execution of Δ_s in which all reconfiguration actions are grouped together and thus could be performed in a single atomic step. To this end, we define $(\tilde{N}, p, q)$ as a triple with $\tilde{N}$ being a configuration in which all $c \in \mathrm{dom}(\tilde{N})$ either are running or blocked; p a sequence of planned reconfiguration actions for Δ_s in the order given by the execution of Δ_s; and q a sequence of non-reconfiguration actions. We say that a component configuration $\tilde{C}$ is *simulated* by a triple $(\tilde{N}, p, q)$, written as $\tilde{C} \preccurlyeq (\tilde{N}, p, q)$, if there exist $\tilde{C}^{(1)}$ and $\tilde{C}^{(0)}$ such that

$$\tilde{C} = \tilde{C}^{(0)} \xleftarrow{q} \tilde{C}^{(1)} \xleftarrow{p} \tilde{N} \ ;$$

note that if $\tilde{C}^{(1)}$ and $\tilde{C}^{(0)}$ exist, they are uniquely determined. The regrouping of reconfiguration actions in a simulating execution is afforded by a transition system $(\tilde{N}, p, q) \overset{a}{\Longrightarrow} (\tilde{N}', p', q')$ on triples defined by

$$\begin{array}{ll}
(\tilde{N}, p, q) \overset{a}{\Longrightarrow} (\tilde{N}', p, q) & \text{if } a \text{ is a non-reconfiguration action} \\
& \quad \text{and } p \text{ contains only RCSTOP actions} \\
& \quad \text{and } \tilde{N} \xrightarrow{a} \tilde{N}', \\
(\tilde{N}, p, q) \overset{a}{\Longrightarrow} (\tilde{N}, p :: a, q) & \text{if } a \text{ is a reconfiguration action,} \\
(\tilde{N}, p, q) \overset{a}{\Longrightarrow} (\tilde{N}, p, q :: a) & \text{if } a \text{ is a non-reconfiguration action} \\
& \quad \text{and } p \text{ contains an action other than RCSTOP.}
\end{array}$$

Proposition 5.4 *Let $\Delta_s = (A, R, \alpha, \rho, \varsigma)$ be a shallow reconfiguration plan, and let the sequence $\tilde{C}_0 \xrightarrow{a_0} \tilde{C}_1 \xrightarrow{a_1} \ldots \xrightarrow{a_{n-1}} \tilde{C}_n$ be an interleaved plan execution of Δ_s. Then there is a sequence $(\tilde{N}_0, p_0, q_0) \overset{a_0}{\Longrightarrow} \ldots \overset{a_{n-1}}{\Longrightarrow} (\tilde{N}_n, p_n, q_n)$ such that $\tilde{C}_k \preccurlyeq (\tilde{N}_k, p_k, q_k)$ for all $0 \le k \le n$ with $\tilde{N}_0 = \tilde{C}_0$, $p_0 = \varepsilon$ and $q_0 = \varepsilon$.*

Proof. First of all, $\tilde{C}_0 \preccurlyeq (\tilde{C}_0, \varepsilon, \varepsilon) = (\tilde{N}_0, p_0, q_0)$. Let the claim hold up to some

$0 \leq k < n$. In order to show that there is an $(\tilde{N}_{k+1}, p_{k+1}, q_{k+1})$ with $(\tilde{N}_k, p_k, q_k) \stackrel{a_k}{\Longrightarrow} (\tilde{N}_{k+1}, p_{k+1}, q_{k+1})$ and $\tilde{C}_{k+1} \preccurlyeq (\tilde{N}_{k+1}, p_{k+1}, q_{k+1})$ we proceed by a case distinction on the action a_k:

If a_k is a non-reconfiguration action and p_k contains an action other than RCSTOP, we may trivially choose $(\tilde{N}_k, p_k, q_k :: a_k)$.

If a_k is a non-reconfiguration action, but p_k only contains RCSTOP actions, we have to provide $\tilde{N}_{k+1}$, $\tilde{C}^{(1)}_{k+1}$, and $\tilde{C}^{(0)}_{k+1}$ such that the following diagram commutes:

$$
\begin{array}{ccccccc}
\tilde{C}_k & = & \tilde{C}^{(0)}_k & \xleftarrow{q_k} & \tilde{C}^{(1)}_k & \xleftarrow{p_k} & \tilde{N}_k \\
\downarrow a_k & & \downarrow a_k & & & & \vdots\, a_k \\
\tilde{C}_{k+1} & = & \tilde{C}^{(0)}_{k+1} & \overset{q_k}{\dashleftarrow} & \tilde{C}^{(1)}_{k+1} & \overset{p_k}{\dashleftarrow} & \tilde{N}_{k+1}
\end{array}
$$

But as p_k only contains RCSTOP actions, q_k is ε; the component conducting a_k is not a parameter of an action in p_k and hence remains unaffected by p_k. Thus $\tilde{N}_{k+1}$ and $\tilde{C}^{(0)}_{k+1} = \tilde{C}^{(1)}_{k+1}$ can be defined as the result of applying a_k to $\tilde{N}_k$ and p_k to $\tilde{N}_{k+1}$.

If a_k is a reconfiguration action, we have to provide $\tilde{C}^{(1)}_{k+1}$ and $\tilde{C}^{(0)}_{k+1}$ such that the following diagram commutes:

$$
\begin{array}{ccccccc}
\tilde{C}_k & = & \tilde{C}^{(0)}_k & \xleftarrow{q_k} & \tilde{C}^{(1)}_k & \xleftarrow{p_k} & \tilde{N}_k \\
\downarrow a_k & & \downarrow a_k & & \vdots\, a_k & & \\
\tilde{C}_{k+1} & = & \tilde{C}^{(0)}_{k+1} & \overset{q_k}{\dashleftarrow} & \tilde{C}^{(1)}_{k+1} & &
\end{array}
$$

All actions in q_k have been invoked for a component in state r or b. Hence all reconfiguration actions applying to a single component only (except RCSTOP) apply to some other state (n, i, s or c). In particular, the rules RCADD, RCINIT, RCSTART and RCKILL are independent of q_k. Rules RCWIRE and RCCOPY do not modify the other components, and do not rely on parts of the component configuration that can be changed by normal rules. For RCSTOPS and RCSTOPF, we have that $q_k = \varepsilon$, since a plan is executed and we would only have actions in q_k once all $c \in R$ are stopped. RCSTATE also only uses components in a reconfiguration state and only uses the state part of their configuration that is not changed by actions in q_k. This leaves two reconfiguration actions that actually interfere with actions in q_k:

- $a_k \equiv$ RCREWIRE(c_1, c_3, r): Any message sending from c_1 over rôle r in q_k will be executed as a message sending to c_3. Using δ^ρ, this is where the messages sent over this rôle are prepended to, and since $|\rho^{-1}(c_3)| \leq 1$ the order is kept the same.
- $a_k \equiv$ RCGETMSG(c_1, c_2, c_3, r): As the messages are prepended to the queue of c_1 and c_1 is in state i and has not yet dequeued any message, the copying does not interfere with any sending action in q_k. If c_2 is the target of a sending action in q_k, ρ will point $c_3.r$ to c_1, which is where the message would have been copied to according to δ^ρ. Component c_3 is not modified.

Thus we can choose $\tilde{C}^{(1)}_{k+1}$ to be the result of applying a_k to $\tilde{C}^{(1)}_k$ and $\tilde{C}^{(0)}_{k+1}$ as the result of applying q_k to $\tilde{C}^{(0)}_{k+1}$. □

This proposition asserts us that plan execution is "perceived as atomic", meaning that it is indistinguishable from an atomic reconfiguration, i.e., from a trace first evolving to $\tilde{N}_n$ by normal actions, then applying the reconfiguration actions of p_n, followed by further normal actions of q_n. Hence, we do not have to stop components that are not scheduled to be stopped in order to reconfigure the system. As discussed before, this is a vital property for the applicability of reconfiguration in large component systems.

6 Verifying Properties in the Presence of Reconfiguration

Verification of properties of component systems that can be reconfigured needs to take into account all possible reconfiguration sequences. For non-atomic reconfiguration, the number of states grows considerably. Due to Prop. 5.4, we are allowed to check only atomic reconfigurations, and be asserted that any interleaved plan execution behaves likewise. To formally show this, we consider state formulas, which are sets of configurations. For a configuration $\tilde{C}$ and a state formula φ, we write $\tilde{C} \models \varphi$ for $\tilde{C} \in \varphi$. We extend this notion to a transition system $\mathscr{T}$ by $\mathscr{T} \models \varphi$ iff $\tilde{C} \models \varphi$ for every reachable state $\tilde{C}$ of $\mathscr{T}$.

Corollary 6.1 *Let $\mathscr{T}_a$ be the transition system of a component system that can get reconfigured by (atomic) executions of a shallow plan Δ_s, and let $\mathscr{T}_i$ be the transition system of the same component system that can get reconfigured by interleaved executions of Δ_s. If $\mathscr{T}_a \models \varphi$ for a state property φ, then $\mathscr{T}_i \models \varphi$.*

Proof. Assume that $\mathscr{T}_i \not\models \varphi$. Then there is a (rooted) path $\tilde{C}_1 \dots \tilde{C}_n$ of $\mathscr{T}_i$ such

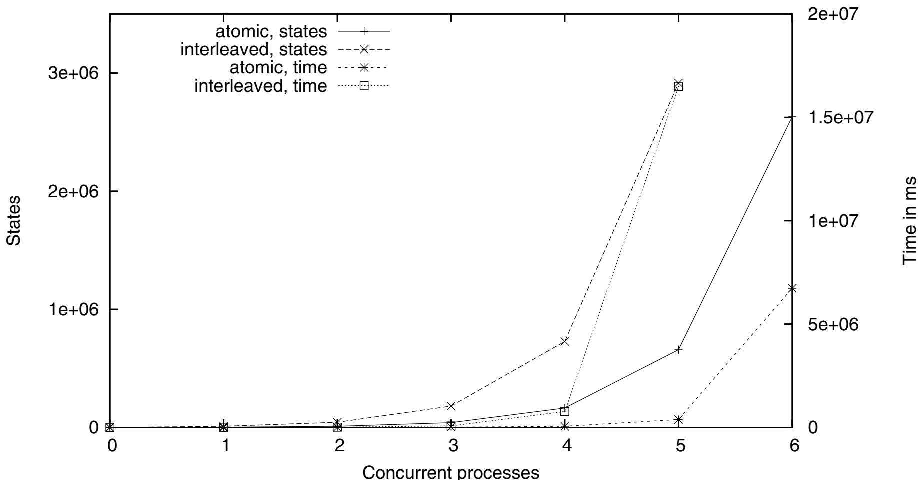


Fig. 5. Atomic vs. interleaved reconfiguration in Maude. 6 concurrent processes for interleaved reconfiguration exceeded the main memory of 16GB.

that $\tilde{C}_n \not\models \varphi$. Since $\mathscr{T}_a$ and $\mathscr{T}_i$ differ only in their reconfiguration, $\tilde{C}_1 \dots \tilde{C}_n$ is an interleaved plan execution of Δ_s. By Prop. 5.4, we can produce a sequence $\tilde{C}'_1 \dots \tilde{C}'_n$ with $\tilde{C}'_n = \tilde{C}_n$, hence $\mathscr{T}_a \not\models \varphi$. □

Doing an atomic step instead of a series of fine-grained steps reduces the state-space, as the combinations of the intermediate reconfiguration states with the states of independent, concurrent components are not produced.

We have implemented the interleaved reconfiguration plan execution and the rules for normal component term execution in Maude [6], and used its search capabilities to verify that the client of the fault-tolerance example cannot observe the reconfiguration, if the new store component provides the same return values as the old ones. Atomic reconfiguration is derived from the interleaved reconfiguration by setting a flag that prohibits the execution of normal rules during a reconfiguration. Fig. 5 shows the size of the state space and the time required to produce it (by searching for an unreachable state, like the client being passed a corrupt value) for atomic and interleaved reconfiguration.

7 Conclusions

We have introduced a framework for concurrent, stateful components that communicate by means of synchronous and asynchronous messages, and a procedure to reconfigure such a stateful system according to a plan such that states and messages are retained. An important property of this procedure is its minimal invasiveness: only those components that need to be removed are actively stopped in the course of reconfiguration execution.

The component framework presented is a precise description of an implementation on top of Java. We found that the assumptions made in the model — e.g., the absence of shared data, in particular the restrictions on synchronous call-backs, — are crucial to handling state-retaining reconfiguration in a real programming language. The locality of the reconfiguration greatly facilitates the planning of reconfiguration, which is already hard enough — it requires careful planning to ensure that a plan can indeed be executed (i.e., all components in R eventually reach $\mathrm{r}, \mathrm{success}$).

However, the proposed component model is but one of many possible, useful models. We expect our results to be applicable to other, more complicated models as well. For example, a model might choose to issue time-stamps to the messages; this would allow to lift the $|\rho^{-1}(c)| \leq 1$ property for shallow reconfiguration plans. Due to the use of reconfiguration plans, we expect our approach to integrate well with reconfiguration planning methods like [5]. Integrating these techniques offers a strong and practicable approach towards the reconfiguration of stateful components.

References

[1] T. Barros, A. Cansado, E. Madelaine, and M. Rivera. Model-checking distributed components: The Vercors platform. *Electron. Notes Theor. Comput. Sci.*, 182:3–16, 2007.

[2] C. Bidan, V. Issarny, T. Saridakis, and A. Zarras. A dynamic reconfiguration service for CORBA. In *International Conference on Configurable Distributed Systems*. IEEE Computer Society, 1998.

[3] T. Bloom. Dynamic module replacement in a distributed programming system. Technical Report MIT/LCS/TR-303, Massachusetts Institute of Technology, 1983.

[4] J. S. Bradbury, J. R. Cordy, J. Dingel, and M. Wermelinger. A survey of self-management in dynamic software architecture specifications. In *Workshop on Self-managed Systems*, pages 28–33. ACM, 2004.

[5] R. Bruni, A. L. Lafuente, and U. Montanari. Hierarchical design rewriting with Maude. In *International Workshop on Rewriting Logic and its Applications*, Electronic Notes in Theoretical Computer Science. Elsevier, 2008. to appear.

[6] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. *All About Maude - A High-Performance Logical Framework: How to Specify, Program, and Verify Systems in Rewriting Logic*, volume 4350 of *Lecture Notes in Computer Science*. Springer-Verlag, 2007.

[7] M. P. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):463–492, 1990.

[8] C. R. Hofmeister. *Dynamic Reconfiguration of Distributed Applications*. PhD thesis, Computer Science Department, University of Maryland, 1993.

[9] F. Irmert, T. Fischer, and K. Meyer-Wegener. Runtime adaptation in a service-oriented component model. In M. Litoiu and H. Giese, editors, *Software Engineering for Adaptive and Self-Managing Systems*, pages 97–104. ACM, 2008.

[10] J. Kramer and J. Magee. The evolving philosophers problem: Dynamic change management. *IEEE Transactions on Software Engineering*, 16(11):1293–1306, 1990.

[11] K.-K. Lau and Z. Wang. A Survey of Software Component Models (2nd ed.). Technical Report CSPP-38, School of Computer Science, The University of Manchester, 2006.

[12] A. S. Lim. Abstraction and composition techniques for reconfiguration of large-scale complex applications. In *International Conference on Configurable Distributed Systems*, page 186. IEEE Computer Society, 1996.

[13] P. K. McKinley, S. M. Sadjadi, E. P. Kasten, and B. H. C. Cheng. A taxonomy of compositional adaptation. Technical Report MSU-CSE-04-17, Department of Computer Science, Michigan State University, 2004.

[14] R. Milner. *Communicating and Mobile Systems: The Pi-Calculus*. Cambridge Univ. Press, 1999.

[15] A. Mukhija and M. Glinz. Runtime adaptation of applications through dynamic recomposition of components. In M. Beigl and P. Lukowicz, editors, *Architecture of Computing Systems*, volume 3432 of *Lecture Notes in Computer Science*, pages 124–138. Springer, 2005.

[16] P. Parizek, F. Plasil, and J. Kofron. Model checking of software components: Combining Java PathFinder and behavior protocol model checker. In *Proc. 30th IEEE/NASA Software Engineering Wsh. (SEW'06)*, pages 133–141. IEEE Computer Society, 2006.

[17] A. Rasche and A. Polze. ReDAC dynamic reconfiguration of distributed component-based applications with cyclic dependencies. In *11th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, pages 322–330. IEEE Computer Society, 2008.

[18] A. Rasche and W. Schult. Dynamic updates of graphical components in the .NET framework. In *Workshop on Selbstorganisierende, Adaptive, Kontextsensitive Verteilte Systeme*, 2007.

[19] A. Rensink. The GROOVE simulator: A tool for state space generation. In J. Pfalz, M. Nagl, and B. Böhlen, editors, *Applications of Graph Transformations with Industrial Relevance*, volume 3062 of *Lecture Notes in Computer Science*, pages 479–485. Springer-Verlag, 2004.

[20] L. A. Tewksbury, L. E. Moser, and P. M. Melliar-Smith. Live upgrade techniques for CORBA applications. In *Third International Working Conference on New Developments in Distributed Applications and Interoperable Systems*, pages 257–272. Kluwer, B.V., 2001.

[21] Y. Vandewoude. *Dynamically updating component-oriented systems*. PhD thesis, Katholieke Universiteit Leuven, 2007.

[22] Y. Vandewoude and Y. Berbers. An overview and assessment of dynamic update methods for component-oriented embedded systems. In H. R. Arabnia and Y. Mun, editors, *International Conference on Software Engineering Research and Practice*, pages 521–527. CSREA Press, 2002.

[23] M. Wegdam. *Dynamic Reconfiguration and Load Distribution in Component Middleware*. PhD thesis, University of Twente, Enschede, The Netherlands, 2003.

[24] M. Wermelinger and J. L. Fiadeiro. Algebraic Software Architecture Reconfiguration. *ACM SIGSOFT Softw. Notes*, 24(6):393–409, 1999.