Using Model-Checking Techniques for Component-Based Systems with Reconfigurations

Jean-Michel Hufflen
FEMTO-ST (UMR CNRS 6174) & University of Franche-Comté
16, route de Gray; 25030 Besançon Cedex; France
jmhuffle@femto-st.fr

Within a component-based approach allowing dynamic reconfigurations, sequences of successive reconfiguration operations are expressed by means of reconfiguration paths, possibly infinite. We show that a subclass of such paths can be modelled by finite state automata. This feature allows us to use techniques related to model-checking to prove some architectural, event, and temporal properties related to dynamic reconfiguration. Our method is proved correct w.r.t. these properties' definition.
Keywords Model checking, finite state automata, component-based approach, checking invariance properties, dynamic reconfiguration paths.

1 Introduction

This article aims to show that some properties related to component-based software with reconfigurations can be proved by implementations based on model-checking techniques. Let us recall that most of component-based systems aim to run for a large period of time, so some components may fail or need to be improved or replaced. That is why dynamic reconfigurations increase the availability and reliability of such systems by allowing their architecture to evolve at runtime. Dynamic reconfigurations of software architectures is an active research topic [1, 3, 18, 19, 20], motivated by practical distributed applications. Such applications are put into action by means of toolboxes such as Fractal [4]. In addition, if we consider systems with high-safety requirements, the verification of *architectural*, *event* and *temporal* properties may be crucial. Some proposals exist, e.g., [12], which focus on the verification of properties related to the behaviour of component-based systems. Within this framework, [9] proposes FTPL[1], a temporal logic for dynamic reconfigurations, including such properties. These properties apply to successive *configurations*—or *component models*—, chaining reconfigurations being modelled by *reconfiguration paths*. Since FTPL is based on first-order predicate logic, such properties are undecidable in general, there only exist partial solutions for proving them.

Within this framework, [16, 17] developed methods that work whilst software is running and may be reconfigured. Therefore we know if a property holds step by step, until the current runtime state. Our method proceeds from a different point of view; our *modus operandi* is more related to the approach of a procedure's developer when such a developer aims to prove its procedure before deploying it and putting it into action. More precisely, we propose a method for verifying such properties, not at runtime, but on a static abstraction of the reconfiguration model, so we aim to ensure that such a property holds before the software is deployed and working, that is, at design-time. In other words, given a reconfiguration path that may be applied when the software is running, we aim to ensure that a property holds if this path is actually applied when the software works. Our method is suitable for finite reconfiguration paths, but also for infinite ones, provided that they can be modelled by finite expressions, that is, by using the '+'

[1] **F**ractal **T**emporal **P**attern **L**ogic.

B. Buhnova, L. Happe, J. Kofroň (Eds.):
Formal Engineering approaches to Software
Components and Architectures (FESCA'15).
EPTCS ??, 2015, pp. 34–47, doi:10.4204/EPTCS.??.6

operator of regular expressions at a final position. In other words, our method applies to finite reconfiguration paths, and also to cycles without continuation. This last condition may appear as restrictive, but in practice, the same sequences are often repeated: a component may be stopped in some circumstances, restarted in other circumstances, and so on. So the repetition of identical reconfiguration sequences seems to us to be interesting in practice, even if they are only a subset of infinite reconfiguration paths. Besides, our implementation is operational, fits the notion of reconfiguration path and opens promising ways about properties related to reconfigurations. In addition, we can prove that this implementation is correct w.r.t. the definitions given in [9]. Section 2 gives some recalls about the component model we use, our operations of reconfiguration, and the temporal logic for dynamic reconfigurations. Of course, most definitions presented in this section come from [9, 10, 11, 16]. Section 3 is devoted to the main outlines of our framework. Then we give some examples of our programs in Section 4 and study the correctness of these implementations w.r.t. the operators defined in Section 2. In this article, we do not examine the implementation of all the operators—given in [14]—but our examples are representative. Section 5 discusses some advantages and drawbacks of our method, in comparison with other approaches. It also introduces future work.

2 Architectural Reconfiguration Model

First we recall how our *component model* is organised. Then we sum up the operations used for reconfiguring an architecture. Last, we make precise operators used in FTPL, the temporal logic used in [9, 10, 11, 16] for dynamic reconfigurations.

2.1 Component Model

Roughly speaking, a component model describes an *architecture* of components. Some simpler components may be subcomponents of a *composite* one, and components may be *linked*. Let $\mathscr{S}$ be a set of *class names*—in the sense used in object-oriented programming—a *component* $\mathscr{C}$ is defined by:

- three pairwise-disjoint sets of *parameters*[2] $P_{\mathscr{C}}$, *input port* names $I_{\mathscr{C}}$, and *output port* names $O_{\mathscr{C}}$;
- the class $t_{\mathscr{C}}$ encompassing the services implemented by the component;
- additional functions to get access to the class of a parameter or port ($\tau_C : P_{\mathscr{C}} \cup I_{\mathscr{C}} \cup O_{\mathscr{C}} \to \mathscr{S}$), or to a parameter's value ($v_{\mathscr{C}} : P_{\mathscr{C}} \to \bigcup \mathscr{S}$);
- the set *sub-c*$_{\mathscr{C}}$ of its subcomponents if the $\mathscr{C}$ component is composite. Of course, the binary relation 'is a subcomponent of' must be a direct acyclic graph.

A composite component cannot have parameters. The *bindings* of ports form a set B of couples of output and input port names, being the same type. Delegation links, between composite component ports and ports of contained components form a set D of couples, too. As an example of a component-based architecture, possible components of an HTTP server are given in Fig. 1.

2.2 Configuration Properties

Example 1 *Looking at Fig. 1's architecture, we can notice that the* `CacheHandler` *component is connected to the* `RequestHandler` *component through their respective ports* `cache` *and* `getCache`. *We can*

[2] Some authors use the term 'attributes' instead. A parameter is related to an internal feature, e.g., the maximum number of messages a component can process.

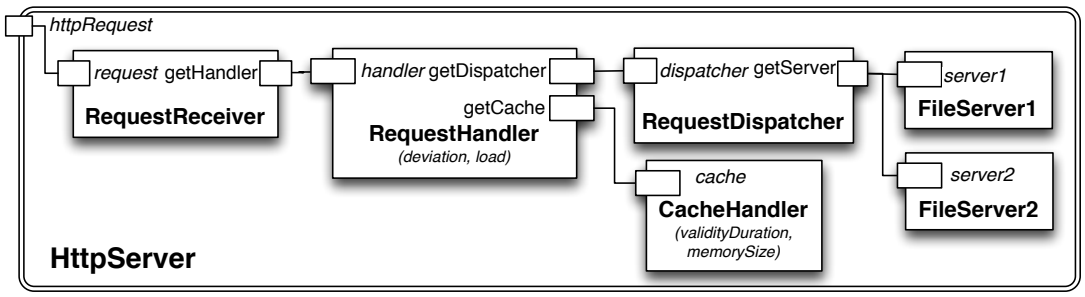

Figure 1: Architecture of an HTTP server [11].

express this configuration property—*so-called* `CacheConnected`—*as follows:*

$$B \ni (\texttt{cache}_{\texttt{CacheHandler}}, \texttt{getCache}_{\texttt{RequestHandler}})$$

In fact, such properties—that may be viewed as *constraints*—are specified using first-order logic formulas over constants ('`true`', '`false`'), variables, sets and functions defined in § 2.1, predicates $(=, \in, \ldots)$, connectors $(\wedge, \vee, \ldots)$ and quantifiers $(\forall, \exists)$. These configuration properties form a set denoted by *CP*.

2.3 Reconfiguration Operations

Primitive reconfiguration operations have been defined, they applied to a component architecture, and the output is a component architecture, too[3]. They are the addition or removal of a component, the addition or removal of a binding, the update of a parameter's value. Let us notice that the result of such an operation is consistent from a point of view related to software architecture: for example, a component is stopped before it is removed, and removing it causes all of its bindings to be removed, too. Another point is that these operations are robust in the sense that they behave like the identity function if the corresponding operation cannot be performed. For example, if you try to remove a component not included in an architecture, the original architecture will be returned. The same if you try to add a component already included in the architecture[4]. As a consequence, these *topological* operations—addition or removal of a component or a binding—are *idempotent*: applying such an operation twice results in the same effect than applying it once. General reconfiguration operations on an architecture are combinations of primitive ones, and form a set denoted by $\mathscr{R}$. The set of *evolution operations* is $\mathscr{R}_{run} = \mathscr{R} \cup \{run\}$ where *run* is an action modelling that all the stopped components are restarted and the software is running.

Definition 2 ([10, 16]) *The operational semantics of component systems with reconfigurations is defined by the labelled transition system* $\mathscr{S} = \langle C, C^0, \mathscr{R}_{run}, \rightarrow, l \rangle$ *where* $C = \{c, c_1, c_2, \ldots\}$ *is a set of configurations,* $C^0 \subseteq C$ *is a set of initial configurations,* $\mathscr{R}_{run}$ *is a finite set of evolution operations,* $\rightarrow \subseteq C \times \mathscr{R}_{run} \times C$ *is the reconfiguration relation, and* $l : C \rightarrow CP$ *is a total function to label each* $c \in C$ *with the largest conjunction of* $cp \in CP$ *evaluated to 'true' over* $\mathscr{R}_{run}$.

[3] They may be viewed as *graph transformations* applied to component models if we consider such models as graphs.

[4] The reason: the *name* of a component—part of its definition—can only identify *one* component.

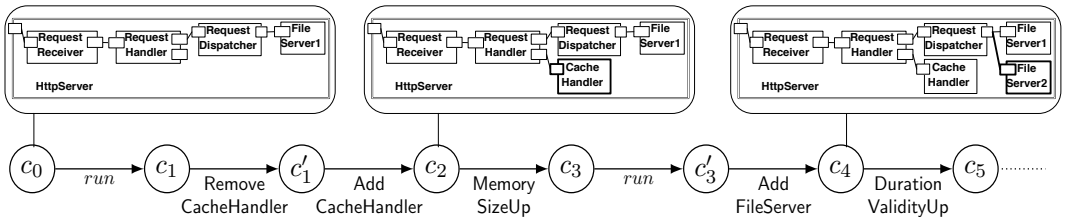

Figure 2: Part of an evolution path of Fig. 1's HTTP server architecture [11].

Let us note $c \xrightarrow{op} c'$ when a target configuration c' is reached from a configuration c by an evolution $op \in \mathscr{R}_{run}$. Given the model $S = \langle C, C^0, \mathscr{R}_{run}, \rightarrow, l \rangle$, an evolution path σ of S is a (possibly infinite) sequence of component models $c_0, c_1, c_2, \ldots$ such that $\forall i \in \mathbb{N}, \exists op \in \mathscr{R}_{run}, c_i \xrightarrow{op} c_{i+1} \in \rightarrow$. We write '$\sigma[i]$' to denote the ith element of a path σ. The notation '$\sigma_i^{\uparrow}$' denotes the suffix path $\sigma[i], \sigma[i+1], \ldots$ and 'σ_i^j' ($j \in \mathbb{N}$) denotes the segment path $\sigma[i], \sigma[i+1], \ldots, \sigma[j-1], \sigma[j]$. An example of reconfiguration path allowing Fig. 1 to be reached from a simpler architecture is given in Fig. 2 (Fig. 1's architecture is labelled by the c_4 configuration).

2.4 Temporal Logic

FTPL contains events from reconfiguration operations, trace properties, and temporal properties, respectively denoted by '*event*', '*trace*', and '*temp*' in the following. Hereafter we only give some operators of FTPL, in particular those used in the implementations we describe. For more details about this temporal logic, see [10, 16]. FTPL's syntax is defined by:

$$\begin{array}{lcl} \langle temp \rangle & ::= & \textbf{after}\ \langle event \rangle\ \langle temp \rangle \mid \textbf{before}\ \langle event \rangle\ \langle trace \rangle \mid \ldots \\ \langle trace \rangle & ::= & \textbf{always}\ cp \mid \textbf{eventually}\ cp \mid \ldots \\ \langle event \rangle & ::= & op\ \textbf{normal} \mid op\ \textbf{exceptional} \mid op\ \textbf{terminates} \end{array}$$

where '*cp*' is a configuration property and '*op*' a reconfiguration operation. Let *cp* in *CP* be a configuration property and c a configuration, c satisfies *cp*, written '$c \models cp$' when $l(c) \Rightarrow cp$. Otherwise, we write '$c \not\models cp$' when c does not satisfy *cp*.

Definition 3 ([10]) *Let σ be an evolution path, the* FTPL *semantics is defined by induction on the form of the formulas as follows*[5]*—in the following, $i \in \mathbb{N}$—:*

- for the events:

$$\begin{array}{lll} \sigma[i] \models op\ \textbf{normal} & \text{if} & i > 0 \wedge \sigma[i-1] \neq \sigma[i] \wedge \sigma[i-1] \xrightarrow{op} \sigma[i] \in \rightarrow \\ \sigma[i] \models op\ \textbf{exceptional} & \text{if} & i > 0 \wedge \sigma[i-1] = \sigma[i] \wedge \sigma[i-1] \xrightarrow{op} \sigma[i] \in \rightarrow \\ \sigma[i] \models op\ \textbf{terminates} & \text{if} & \sigma[i] \models op\ \textbf{normal} \vee \sigma[i] \models op\ \textbf{exceptional} \end{array}$$

- for the trace properties:

$$\begin{array}{lll} \sigma \models \textbf{always}\ cp & \text{if} & \forall i : i \geq 0 \Rightarrow \sigma[i] \models cp \\ \sigma \models \textbf{eventually}\ cp & \text{if} & \exists i : i \geq 0 \Rightarrow \sigma[i] \models cp \end{array}$$

[5] For a complete definition including all the operators, see [10].

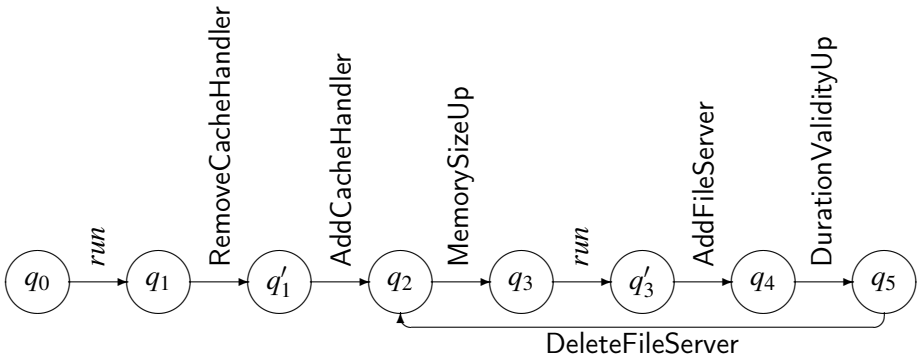

Figure 3: Reconfiguration path viewed as a finite state automaton.

- for the temporal properties:

$$\sigma \models \textbf{after } \textit{event temp} \quad \text{if} \quad \forall i : i \geq 0 \wedge \sigma[i] \models \textit{event} \Rightarrow \sigma_i^{\uparrow} \models \textit{temp}$$
$$\sigma \models \textbf{before } \textit{event trace} \quad \text{if} \quad \forall i : i > 0 \wedge \sigma[i] \models \textit{event} \Rightarrow \sigma_0^{i-1} \models \textit{trace}$$

Example 4 *If we consider the evolution path of Fig. 2 again, we can now express that after calling the* AddCacheHandler *reconfiguration operation, the* `CacheHandler` *component is always connected to the* `RequestHandler` *component—*`CacheConnected` *is the configuration property defined in Example 1—:*

after AddCacheHandler **normal always** `CacheConnected`

3 Our Method's Main Outlines

3.1 Basic Idea

Our basic idea is that a finite reconfiguration path may be viewed as a particular case of a *finite state automaton*, more precisely, a kind of *Büchi automaton*. This property still holds if an infinite reconfiguration path may be expressed using the '+' operator of regular expressions at a final position. As an example, let us consider the following reconfiguration path, related to Fig. 2:

run RemoveCacheHandler AddCacheHandler
(MemorySizeUp *run* AddFileServer DurationValidityUp DeleteFileServer)+

it can be modelled by the automaton pictured in Fig. 3 (before cycling, the states $q_0, q_1, q'_1, \ldots, q_5$ have been respectively named in connection to the successive component models $c_0, c_1, c'_1, \ldots, c_5$. Let us recall that a finite state automaton $\mathscr{A}$ is defined by a set Q of *states*, a set L of *transition labels*, and a set $T \subseteq Q \times L \times Q$ of *transitions*. Like in Definition 3 for systems with reconfigurations, there exists a function $l : Q \rightarrow CP$, which labels each q state with the largest conjunction of $cp \in CP$ evaluated to 'true' for the q state.

Within this framework, a state of such an automaton modelling a reconfiguration path is a component model, initial or got by means of successive reconfiguration operations—primitive or built by chaining primitive operations—or '*run*' operations. A transition consists of applying such an evolution operation. Of course, such automata are particular cases: there is only one initial state, and only one transition can be applied from a state. More formally, the T set of such an $\mathscr{A}$ automaton satisfies:

$$\forall q, q_0, q_1 \in Q, \forall l_0, l_1 \in L, (q, l_0, q_0) \in T \wedge (q, l_1, q_1) \in T \Rightarrow l_0 = l_1 \wedge q_0 = q_1$$

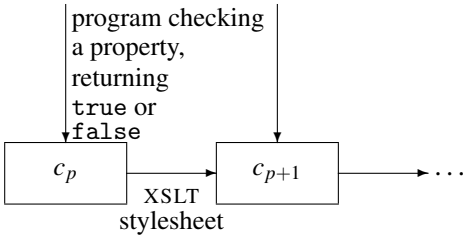

Figure 4: Our organisation.

In the following, we use the notations '$succ_{\mathscr{A}}(q) = q_0$' or '$q \overset{succ_{\mathscr{A}}}{\mapsto} q_0$' for the q_0 state reached from the q state by a unique transition: $succ_{\mathscr{A}}(q) = q_0 \Leftrightarrow \exists! l_0 \in L, (q, l_0, q_0) \in T$. If a reconfiguration path is finite, the corresponding automaton has a final state. Otherwise (like in the example above), there is no final state in the sense that no transition can be performed. If a state of such an automaton is reached several times—e.g., the q_2 state in Fig. 3, reached after q'_1 and q_5—considering that the whole system is back to a previous state is not exact, because some parameters can have been updated: this is the case in Fig. 3's example, about the memory's size and duration validity. As a consequence, some properties related to components' parameters may not hold. We will go back on this point at the end of § 4.2.

3.2 *Modus Operandi*

We use several programming languages within our framework. At first glance, this choice introduces some complexity, but in reality, each language is used suitably. Fig. 4 shows how tasks are organised within our architecture—$(c_p)_{p\in\mathbb{N}}$ being successive component models. In our implementation, the ADL[6] we use for our component models is TACOS+/XML [13]. This language using XML[7]-like syntax is comparable with other ADLs, so from a theoretical point of view, we could use any ADL, another example being Fractal/ADL [4], but we mention that the organisation of TACOS+/XML texts make very easy the programming of primitive reconfiguration operations mentioned in § 2.3, that is why we chose this ADL. Reconfigurations operations are implemented using XSLT[8]: the input and output are TACOS+/XML files. When the software is running, only one component model is in use, so that may be viewed as the identity function applied to a component model.

A short example of such a TACOS+/XML file is given in Fig. 5. In our implementation, configuration properties are expressed using XQuery programs [26], returning 'true' or 'false'[9], as exemplified in Fig. 6 about the `CacheConnected` property. It is well-known that XML dialects are very suitable for specifying architectures—most of ADLs use this syntax—and XSLT/XQuery are very appropriate for operations modelling reconfigurations and property checks. Chaining reconfigurations and property checks is expressed by an implementation of automata. There is no difficulty about the implementation of recon-

[6] **A**rchitecture **D**efinition **L**anguage.

[7] e**X**tensible **M**arkup **L**anguage.

[8] e**X**tensible **S**tylesheet **L**anguage **T**ransformations, the language of transformations used for XML documents [25]. Let us note that if another ADL is used within a project, there exist XSLT programs giving equivalent descriptions in TACOS/XML [13]. In particular, that is the case for Fractal/ADL.

[9] Of course, other choices are possible, in particular XSLT stylesheets. Our point of view: XQuery programs are more concise, what seems to us to be interesting for verifying architectural properties. Besides, our XQuery programs return 'true' or 'false', but we could easily modify them in order to output a counter-example if a property does not hold.

```
<tacos:components ...> ...
  <tacos:component-specifications>
    <tacos:composite-component id="HttpServer" path="HttpServer">
      <tacos:port ref="Trequest" role="server" name="httpRequest"/> ...
      <tacos:refers-to ref="RequestReceiver"/>
      <tacos:refers-to ref="RequestHandler"/> ...
    </tacos:composite-component>
    <tacos:component id="RequestReceiver" path="HttpServer/RequestReceiver">
      <tacos:port ref="Trequest" role="server" name="request"/> ...
    </tacos:component>
    <tacos:component id="RequestHandler" path="HttpServer/RequestHandler">
      <tacos:port ref="Thandler" role="server" name="handler"/> ...
      <tacos:attributes signature="RequestHandlerAttributes">
        <tacos:attribute name="deviation" value="50"/> ...
      </tacos:attributes> ...
    </tacos:component> ...
  </tacos:component-specifications> ...
  <tacos:binding-specifications>
    <tacos:binding from="request" to="httpRequest" server="RequestReceiver"
                   client="HttpServer"/>
    <tacos:binding from="handler" to="getHandler" server="RequestHandler"
                   client="RequestReceiver"/> ...
  </tacos:binding-specifications>
</tacos:components>
```

Figure 5: Example of a component architecture described by means of TACOS+/XML.

figuration operations and property checks, so the descriptions put hereafter concern the part implemented by means of automata.

3.3 Types Used

In this paper, we describe our checking functions at a high level. Hereafter we make precise the types used, in order to ease the reading of our functions. The formalism we used is close to type definitions in strong typed functional programming languages like Standard ML [22] or Haskell [21]. Of course, we assume that some types used hereafter—e.g., '`bool`', '`int`'—are predefined.

As abovementioned, an evolution operation is either the identity function, which expresses that the software is running, or a reconfiguration operation, which is implemented by applying an XSLT stylesheet to an XML document and getting the result as another XML document. At a higher-level, such an evolution operation may be viewed as a function which applies to a component model and returns a component model. Likewise, checking a property may be viewed as a function which applies to a component model and returns a boolean value. Assuming that the `component-model` type has already been defined, we introduce these two function types as:

type `evolution-operation = component-model` $\rightarrow$ `component-model`
type `check-property = component-model` $\rightarrow$ `bool`

Let `state` be the type used for a state of our automata, starting from such a state and a configuration—component model—is expressed by the following type:

type `path-check = state` $\times$ `component-model` $\rightarrow$ `bool`

This `path-check` type is used within:

function `check-after :` `evolution-operation` $\times$ `path-check` $\rightarrow$ `path-check`
function `check-always :` `check-property` $\rightarrow$ `path-check`

```
(:  Some declarations omitted. '$filename' is the XML file analysed.  :)
some $binding as element(tacos:binding) in
  doc($filename)/tacos:components/tacos:binding-specifications/tacos:binding
  satisfies
    data($binding/@from) eq "cache" and data($binding/@to) eq "getCache" and
    data($binding/@server) eq "cacheHandler" and data($binding/@client) eq "requestHandler"
```

Figure 6: The **CacheConnected** property expressed using XQuery.

In other words, `check-always(`*check-p*`)(`q`,`c`)` applies the *check-p* function along the q state, its successor, and so on, starting from the c component model. The result of this expression is a boolean value. As soon as applying the *check-p* function yields 'false', the process stops and the result is 'false'. Likewise, `check-after(`e`,`*check-f*`)(`q`,`c`)` also starts from the q state and the c component model; it applies the *check-f* function as soon as the e event is detected as a transition of the automata. The property related to the *check-f* function is to be checked for all the component models resulting from the application of the successive transitions. As a more complete example, the translation of the formula '**after** e **always** cp'—where e is an event and cp a configuration property—is `check-after(`e`,check-always(`cp`))`, which is a function that applies on a path, starting from a state and component model. The process starts from the initial state of the automaton. Of course, there are similar declarations for the functions `check-before` and `check-eventually` (cf. § 2.4).

3.4 Ordering States of Automata

In this section, we introduce some notions related to our automata—they do not hold about general automata—and used in the following. The states of our automata modelling reconfiguration paths can be ordered with respect to the transitions performed before cycling. Let $\mathscr{A}$ be such an automaton, if q and q' are two states of $\mathscr{A}$:

$$q < q' \overset{\text{def}}{\Longleftrightarrow} \exists (q_1, \ldots, q_n), q \overset{\texttt{succ}_{\mathscr{A}}}{\mapsto} q_1 \overset{\texttt{succ}_{\mathscr{A}}}{\mapsto} \cdots \overset{\texttt{succ}_{\mathscr{A}}}{\mapsto} q_n \overset{\texttt{succ}_{\mathscr{A}}}{\mapsto} q' \text{and } q, q_1, \ldots, q_n, q' \text{ are pairwise-different.}$$

The notation '$q \leq q'$' stands for '$q < q' \vee q = q'$'. The only transition which does not satisfy this property is the transition going back to a state already explored, it starts from the state denoted by $\texttt{q-max}_{\mathscr{A}}$. If we consider the $\mathscr{A}_0$ automaton pictured at Fig. 3, $q_0 < q_1 < q'_1 < q_2 < q_3 < q'_3 < q_4 < q_5 = \texttt{q-max}_{\mathscr{A}_0}$.

4 Our Method

4.1 Functions

Our main idea is quite comparable to the *modus operandi* of a model-checker when it checks the successive states of an automaton in the sense that we mark all the successive functions of a reconfiguration path. The possible values of such a mark are:

`unchecked` the initial mark for the steps not yet explored within a reconfiguration path;

`again` if a universal property (for all the members of a suffix path) is being checked, it must be checked again at this step if it is explored again;

```
check-after(e,check-f)(q,c) ⟶
  if mark(q) == again then true
  else     // mark(q) == unchecked
    mark(q) ⟵ again ; c0 ⟵ t(q)(c) ;
    if t(q) == e then check-f(succ_A(q),c0) else check-after(e,check-f)(succ_A(q),c0)
    end if
  end if
end

check-always(check-p)(q,c) ⟶
  check-p(c) ∧ if mark(q) == checked then true
               else     // mark(q) == unchecked ∨ mark(q) == again
                 mark(q) ⟵ checked ; check-always(check-p)(succ_A(q),t(q)(c))
               end if ;
end
```

Figure 7: Checking properties: two implementations.

`checked` the property has already been checked, and no additional check is needed if this step is explored again.

So, such an automaton modelling a reconfiguration path is pre-processed and its states are marked as `unchecked`. The mark of a q state is denoted by `mark(`q`)`. The transition label starting from such a state is denoted by `t(`q`)`, let us recall that such a transition is the `evolution-operation` type, so it can be applied to a component model c to get the next component model `t(`q`)(`c`)`.

We give two implementations of checking properties in Fig. 7: the functions `check-after` and `check-always`. We use a high-level functional pseudo-language, except for updating marks, which is done by means of side effects. A complete implementation is available at [14], including other features of FTPL, with similar programming techniques and similar methods for proving the termination of our functions and the correctness w.r.t. the definitions given in [9, 10].

4.2 Implementations' Correctness

4.2.1 Termination

Proposition 5 *The function* `check-after` *terminates.*

Let q_0 be the initial state of our automaton, a principal call of the `check-after` function is:

$$\texttt{check-after}(e,\textit{check-f})(q_0,c)$$

where e is an event, *check-f* a check function being `path-check` type, c a component model. Recursive calls of this function satisfy the invariant:

$$\forall q_j : q_0 \leq q_j < q_i, \texttt{mark}(q_j) = \texttt{again}$$

when it is applied to the q_i state. Let $q_k = \texttt{succ}_{\mathscr{A}}(q_i)$. If $q_i < q_k$, the invariant holds. If $q_i = \texttt{q-max}_{\mathscr{A}}$, then $\forall q_j : q_0 \leq q_j \leq \texttt{q-max}_{\mathscr{A}}, \texttt{mark}(q_j) = \texttt{again}$, that is, the next recursive call applies to a state whose mark is `again`. Such a call terminates.

Proposition 6 *The function* `check-always` *terminates.*

This termination proof is similar: a pass is performed by the `check-always` function, but this pass may start after the beginning of a cycle, and the cycle may have to be entered a second time. Globally, two passes may be needed for an expression such that `check-after`$(e,$`check-always`$(cp))$. Before reaching the end of a cycle, the invariant is:

$$\forall q_j : q_0 \leq q_j < q_i, \texttt{mark}(q_j) = \texttt{checked} \vee \texttt{mark}(q_j) = \texttt{again}$$

when the `check-always` function is applied to the q_i state. Roughly speaking, when a cycle is performed, the mark has been set either to `again`, in which case the property has to be checked again, or to `checked`, in which case our function concludes that the temporal property is true. If the mark has been set to `again`, it means that the checking of the temporal property '**always** *cp*' had not begun yet; for example, if we were processing the '**after**' part of '**after** *e* **always** *cp*'. If re-entering a cycle is needed, the invariant is:

$$\forall q_j : \texttt{succ}_{\mathscr{A}}(\texttt{q-max}_{\mathscr{A}}) \leq q_j < q_i, \texttt{mark}(q_j) = \texttt{checked}$$

q_i being the current state. Let us recall that $\texttt{succ}_{\mathscr{A}}(\texttt{q-max}_{\mathscr{A}})$ is the first state of the cycle of the automaton. If the current state reaches the greatest state w.r.t. the '$<$' relation among states, the following recursive call of `check-always` is performed with the situation:

$$\forall q_j : \texttt{succ}_{\mathscr{A}}(\texttt{q-max}_{\mathscr{A}}) \leq q_j < \texttt{q-max}_{\mathscr{A}}, \texttt{mark}(q_j) = \texttt{checked}$$

that is, the `check-always` function terminates at this next call.

4.2.2 Correctness

Concerning the `check-after` function, let us examine the definition of the **after** temporal property: $\sigma \models$ **after** *e temp* if $\forall i : (i \geq 0 \wedge \sigma[i] \models e \Longrightarrow \sigma_i^{\uparrow} \models temp)$—where σ is a reconfiguration path, *e* an event, *temp* a temporal property. It is sufficient to check this property on the first occurrence of the *e* event in the transitions handled by our `check-after` function. The set I of the i integers such that $i \geq 0 \wedge \sigma[i] \models e$ is a subset of $\mathbb{N}$. Since I is a subset of a well-ordered set, it has a smallest element i_0. So we have $\sigma_{i_0}^{\uparrow} \models temp$ and $\forall i \in I, i_0 \leq i$. As a consequence, $\sigma_{i_0}^{\uparrow} \models temp \Longrightarrow \sigma_i^{\uparrow} \models temp$ for each element i of I. Let us consider a call `check-after`(*e*,*check-f*)(q_i,c_i), where c_i is the component model we got in the q_i state, and the following invariant—let us recall that q_0 is the initial state—:

$$\forall q_j : q_0 \leq q_j < q_i, c_j \not\models e$$

holds. If $\texttt{t}(i) = e$, we check the property implemented by *check-f* from the q_i state of the automaton, corresponding to the suffix of the reconfiguration path starting at the q_i state—that is, $\sigma_i^{\uparrow}$ if states are indexed by natural numbers—which is correct w.r.t. the specification. If $\texttt{t}(i) \neq e$, the `check-after` function is recursively called and the invariant holds. By Proposition 5, if such a call of the `check-after` function terminates with `mark(`q_i`) = again`, that means that all the automaton's states have been marked `again`. Such a mark is put whenever the *e* event is not found. Consequently, this event does not appear and the result is the 'true' value.

The case of the `check-always` function is more subtle. Let us recall that after performing a cycle, the mark is set either to `again`, in which case the property has to be checked again, or to `checked`, in which case our function concludes that the temporal property is true. This is not true for any reconfiguration operation, but holds for a large subset. Let us assume that we have processed a reconfiguration operation *op* on a *c* configuration. If we process *op* again after cycling throughout our reconfiguration operations,

the current component model may be different from c. In other words, the system is not necessarily in the same state. A simple counter-example: let us consider Fig. 5 and a reconfiguration operation $op_{\texttt{deviation++}}$ which increments the `deviation` attribute of the `RequestHandler` component. If this operation is repeated and the property to be checked is '**always** `deviation < 100`', this property will be true at the first pass but will fail after some iterations. Now let us consider the reconfiguration operation $op_{\texttt{deviation} \leftarrow \texttt{99}}$ which sets this `deviation` attribute to a new value, less than 100. If this reconfiguration operation is repeated, the property '`deviation < 100`' will always hold. For our purpose, the difference between the two operators $op_{\texttt{deviation++}}$ and $op_{\texttt{deviation} \leftarrow \texttt{99}}$ is that the latter is idempotent, not the former. So using a cycle whose global composition of all the reconfiguration operations is idempotent is sufficient in order for our `check-always` function to behave correctly w.r.t. the specification of the '`always`' operator. It is necessary for checking any property without further hypotheses. But—as an example—another approach is accurate if properties to be checked do not deal with parameters' values. The condition of correctness is the same: the global configuration of all the reconfiguration operations of the cycle must be idempotent, but reconfiguration operations dealing with parameters can be ignored. Let us recall that most reconfiguration operations introduced in § 2.3 are idempotent. For example, let us recall that if we try to remove a component not included in a configuration or add a component already included, this configuration is unchanged. Consequently, applying this operation once or more causes the same effect. Similar remarks can be done for the addition of a component, the addition or removal of a binding. In general, the composition of idempotent operations is not necessarily an idempotent function. In our case, we can show the composition of *topological* operations is idempotent. To give a sketch of the proof, let us mention that for two idempotent functions $f, g : X \to X$ (X being a set), if $g \circ f = f \circ g$, then $g \circ f$ is idempotent, too. In fact:

$$\begin{aligned} (g \circ f) \circ (g \circ f) &= g \circ (f \circ g) \circ f \\ &= g \circ g \circ f \circ f \\ &= g \circ f \end{aligned}$$

The complete proof is tedious, because many combinations are to be examined. Some pairs of topological operations yield the identity functions when composed, other pairs are commutative w.r.t. the '$\circ$' composition operation. Examples coming from Fig. 3 are the transitions labelled by the events AddFileServer and DeleteFileServer, which yield the identity function, whereas the transitions labelled by AddFileServer and DurationValidityUp are commutative.

5 Discussion and Future Work

Within the framework sketched at § 3.2, our automata have been implemented using the Scheme programming language [24]. The complete implementation can be found in [14], and [15] describes our functions in a way closer to this implementation, giving more details about the Scheme structures we used. The descriptions of this paper allow us to be more related to a theoretical model, and to emphasise that our method is close to algorithms based on marks and used in model-checking, e.g., [6, 7, 23]. Our implementation uses Scheme *streams*[10] for reconfiguration paths and includes the fact that a reconfiguration path may be infinite, but only cycles without continuation are processed as shown in § 4. In practice, our functions are called with an additional argument for the maximum number of iterations along a reconfiguration path. If this number is reached, our functions return the part of the path which

[10] That is, *potentially* infinite lists, as implemented within lazy functional programming languages.

is not explored yet and the component model reached, so end-users can launch the process again if they wish. In this case, we do not perform complete checking, but only *bounded* checking. With our implementation, we experienced the property given at Example 4 and the reconfiguration path pictured at Fig. 3; the result is 'true', as expected. We also experienced some variants: for example, if cycling is performed from the q_5 state to the q'_1 state (cf. Fig. 3), the result is 'true', too, and only the first occurrence of the AddCacheHandler event is different from the identity function. As a second variant, if cycling is done from the q_5 state to the q_1 state, the result is 'false': when the states q_1 and q'_1 are explored by the `check-always` function, the property CacheConnected must be checked and it fails about the q'_1 state; however, let us notice that such check after the q_2 state are not performed if this state is reached after cycling from the q_5 state. Practically we have been able to carry out all the examples of [10]; these tests confirmed results expressed theoretically in that article. Other tests based on reconfigurations of a location component[11] are successful. From a theoretical point of view, we explore as few states as possible. In practice, our programs' efficiency is good, in despite of the communications among several programming languages. In addition, we plan to study implementation techniques in order to improve efficiency. We also plan to apply our technique to large-sized case studies.

The idea of modelling reconfiguration paths by means of automata seems to us to be very promising, and we plan to go thoroughly into more expressive cases. In this paper, we are not interested in the *reasons* of reconfigurations: most often they are implemented by means of *policies* in systems like Fractal, some examples can be found in [8]. Reconfiguration operations may be viewed as operations solving unexpected situations, but most of these situations are planned by policies and can be simulated. We plan to enlarge our language of reconfiguration paths, in order to encompass policies. Likewise, we plan to be able to express *reconfiguration alternatives*. In other words, we plan to build more ambitious automata from more expressive reconfiguration paths, provided that we can check interesting properties. This should lead us to propose new algorithms, but also to a new version of the temporal operators given in § 2.4. The operators given in [10, 16] refer to a *linear-time* logic, whereas reconfiguration alternatives should be based on *branching-time* logic. Explaining this difference is easy: [10, 16] observe a process in progress, at runtime, whereas reconfiguration alternative are *possible futures* we explore before the software is deployed and put into action. Another idea could be to move the Model Driven Engineering technical space [2], who would provide more expressive power about model transformations. Other approaches are closer to a semantic level: for example, [18] models reconfiguration operations by means of graph rewriting and uses formal verification techniques along graphs to check properties related to reconfigurations. A comparable approach is followed in [17]. As another example, [5] defines a calculus allowing policies. Our approach concerns a more syntactic level because our reconfiguration operations apply to component models, and result in other component models. Likewise, we assume that the properties we check can be verified syntactically on component models: that is the case, at least for topological properties. Our approach does not cover all cases of reconfiguration paths, but in practice, the same sequences are often repeated, as mentioned in the introduction.

6 Conclusion

As mentioned above, the starting notions, recalled in § 2, come from [9, 10, 11, 16]. Our contribution is our checking method explained in § 3. Practically, it has been actually implemented. Theoretically, we have shown that our *modus operandi* is correct w.r.t. the definitions. Our contribution also includes the relationship between properties related to reconfiguration operations and techniques used within model-

[11] This example is also used in [13, 17].

checking. We think that using such model-checking techniques for properties related to component-based software with possible reconfigurations is an open way to interesting experiments, theoretically as well as practically. Our bridge between reconfiguration operations and model-checking techniques shows that tools developed for the analysis of systems can be reused for the analysis of reconfiguration by simulation of reconfiguration paths. It is well-known that model-checking techniques can validate a model of a system, not the system itself. So does our method: as mentioned in the introduction, *we do not run* components: we only perform a static analysis at design-time. Our approach cannot replace methods applied at runtime [17, 18], when the software has already been deployed and is working, but we think that our method can provide some help at design-time.

Acknowledgements

I am grateful to Olga Kouchnarenko and Arnaud Lanoix, who kindly permitted me to use Figs. 1 & 2. Many thanks to the anonymous referees, who suggested me constructive improvement.

References

[1] Robert B. Allen, Rémi Douence & David Garlan (1998): *Specifying and Analyzing Dynamic Software Architectures*. In E. Astesiano, editor: *Proc.* FASE *1998*, LNCS 1382, Springer, pp. 21–37, doi:10.1007/BFb0053581.

[2] Jean Bézivin (2006): *Model Driven Engineering: an Emerging Technical Space*. In Ralf Lämmel, Jo ao Saraiva & Joost Visser, editors: *International Summer School* GTTSE *2005, revised papers*, LNCS 4143, Springer, Braga, Portugal, pp. 36–64, doi:10.1007/11877028_2.

[3] Marius Bozga, Mohamad Jaber, Nikolaos Maris & Joseph Sifakis (2012): *Modelling Dynamic Architectures Using Dy-BIP*. In Thomas Gschwind, Flavio De Paoli, Volker Gruhn & Matthias Book, editors: *Proc.* SC *2012*, LNCS 7306, Springer, pp. 1–16, doi:10.1007/978-3-642-30564-1_1.

[4] Éric Bruneton, Thierry Coupaye, Matthieu Leclercq, Vivien Quéma & Jean-Bernard Stefani (2006): *The Fractal Component Model and its Support in Java*. *Software Practice and Experience, special issue on Experiences with Auto-adaptive and Reconfigurable Systems* 36(11-12), pp. 1257–1284, doi:10.1002/spe.767.

[5] Roberto Bruni & Ivan Lavanese (2006): PRISMA*: a Mobile Calculus with Parametric Synchronization*. In Ugo Montanari, Don Sannella & Roberto Bruni, editors: *Proc.* TGC *2006*, LNCS 4661, Lucca, pp. 132–149, doi:10.1007/978-3-540-75336-0_9.

[6] Edmund M. Clarke, E. Allen Emerson & A. Prasad Sistla (1986): *Automatic Verification of Finite-State Concurrent System Using Temporal Logic Specifications*. ACM *Transactions on Programming Languages and Systems* 8(2), pp. 244–263, doi:10.1145/5397.5399.

[7] Edmund M. Clarke, Orna Grumberg & David E. Long (1994): *Verification Tools for Finite-State Concurrent Systems*. In Jacobus Willem de Bakker, Willem-Paul de Roever & Grzegorz Rozenberg, editors: *A Decade of Concurrency, Proc.* REX *School/Symp.*, LNCS 803, Springer-Verlag, Noordwijkerhout, The Netherlands, pp. 124–175, doi:10.1007/3-540-58043-3_19.

[8] Julien Dormoy & Olga Koucharenko (2010): *Event-Based Adaptation Policies for Fractal Components*. In: *Proc.* AICCSA *2010*, IEEE Computer Society Press, Hammamet, Tunisia, pp. 1–8, doi:10.1109/AICCSA.2010.5586944.

[9] Julien Dormoy, Olga Kouchnarenko & Arnaud Lanoix (2010): *Using Temporal Logic for Dynamic Reconfigurations of Components*. In Luís Soares Barbosa & Markus Lumpe, editors: *Proc.* FACS *2010*, Guimaraes, Portugal, pp. 200–217, doi:10.1007/978-3-642-27269-1_12.

[10] Julien Dormoy, Olga Kouchnarenko & Arnaud Lanoix (2011): *Runtime Verification of Temporal Patterns for Dynamic Reconfigurations of Components*. In Farhad Arbab & Peter Csaba Ölveczky, editors: *Proc.* FACS *2011*, *LNCS* 7253, Oslo, Norway, pp. 115–132, doi:10.1007/978-3-642-35743-5_8.

[11] Julien Dormoy, Olga Kouchnarenko & Arnaud Lanoix (2012): *When Structural Refinement of Components Keeps Temporal Properties over Reconfigurations*. In Dimitra Giannakopoulou & Dominique Méry, editors: *Proc.* FM *2012*, *LNCS* 7436, pp. 171–186, doi:10.1007/978-3-642-32759-9_16.

[12] Yliès Falcone, Mohamad Jaber, Thanh-Hung Nguyen, Marius Bozga & Saddek Bensalem (2011): *Runtime Verification of Component-Based Systems*. In Gilles Barthe, Alberto Pardo & Gerardo Schneider, editors: *Proc.* SEFM *2011*, *Lecture Notes in Computer Science* 7041, Springer, Montevideo, Uruguay, pp. 204–220, doi:10.1007/978-3-642-24690-6_15.

[13] Jean-Michel Hufflen (2013): *A Framework for Handling Non-Functional Properties within a Component-Based Approach*. In José Luiz Fiadero, Zhiming Liu & Jiyun Xue, editors: *Proc.* FACS *2013*, LNCS 8348, Nánchāng, China, pp. 196–214, doi:10.1007/978-3-319-07602-7_13.

[14] Jean-Michel Hufflen (2014): *Checking Properties of Reconfigurable Component-Based Systems—The Programs*. `http://lifc.univ-fcomte.fr/home/~jmhufflen/texts/tacos-plus/properties.html`.

[15] Jean-Michel Hufflen (2014): *A Method for Checking Properties of Component-Based Systems with Reconfigurations*. Working paper.

[16] Olga Kouchnarenko & Jean-François Weber (2013): *Adapting Component-Based Systems at Runtime via Policies with Temporal Patterns*. In José Luiz Fiadeiro, Zhiming Liu & Jinyun Xue, editors: *Proc.* FACS *2013*, LNCS 8348, Springer, Nánchāng, China, pp. 234–253, doi:10.1007/978-3-319-07602-7_15.

[17] Olga Kouchnarenko & Jean-François Weber (2014): *Decentralised Evaluation of Temporal Patterns over Component-Based Systems at Runtime*. In Ivan Lanese & Éric Madelaine, editors: *Proc.* FACS *2014*, Bertinoro, Italy, pp. 108–126, doi:10.1007/978-3-319-15317-9_7.

[18] Christian Krause, Ziyan Maraikar, Alexander Lazovik & Farhad Arbab (2011): *Modeling Dynamic Reconfigurations in Reo Using High-Level Replacement Systems*. SCP 76, pp. 23–36, doi:10.1016/j.scico.2009.10.006.

[19] Arnaud Lanoix & Olga Kouchnarenko (2014): *Component Substitution through Dynamic Reconfigurations*. In Barbara Buhnova, Lucia Happe & Jan Kofron, editors: *Proc.* FESCA *2014*, EPTCS 147, Grenoble, France, pp. 32–46, doi:10.4204/EPTCS.147.3.

[20] Marc Léger, Thomas Ledoux & Thierry Coupaye (2010): *Reliable Dynamic Reconfigurations in a Reflective Component Model*. In Lars Grunske, Ralf Reussner & Frantisek Plasil, editors: *Proc.* CBSE *2010*, LNCS 6092, Springer, pp. 74–92, doi:10.1007/978-3-642-13238-4_5.

[21] Simon Marlow (2010): *Haskell 2010 Language Report*. `https://www.haskell.org/onlinereport/haskell2010/`.

[22] Lawrence C. Paulson (1996): ML *for the Working Programmer*, 2 edition. Cambridge University Press, doi:10.1017/CBO9780511811326.

[23] Jean-Pierre Queille & Joseph Sifakis (1982): *Specification and Verification of Concurrent Systems in* CESAR. In M. Dezani-Cianaglini & Ugo Montanari, editors: *Proc. 5th International Symposium on Programming*, LNCS 137, Turin, Italy, pp. 337–351, doi:10.1007/3-540-11494-7_22.

[24] Michael Sperber, William Clinger, R. Kent Dybvig, Matthew Flatt, Anton van Straaten, Richard Kelsey, Jonathan Rees, Robert Bruce Findler & Jacob Matthews (2007): *Revised^6 Report on the Algorithmic Language Scheme*. `hhttp://www.r6rs.org`.

[25] W3C (2007): XSL *Transformations (*XSLT*). Version 2.0*. `http://www.w3.org/TR/2007/WD-xslt20-20070123`. W3C Recommendation. Edited by Michael H. Kay.

[26] W3C (2008): *XQuery 1.1*. `http://www.w3.org/TR/xquery-11-20081203`. W3C Working Draft. Edited by Don Chamberlin and Jonathan Siméon.