Checking Properties along Multiple Reconfiguration Paths for Component-Based Systems

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

Reconfiguration paths are used to express sequences of successive reconfiguration operations within a component-based approach allowing dynamic reconfigurations. We use constructs from regular expressions—in particular, alternatives—to introduce multiple reconfiguration paths. We show how to put into action procedures allowing architectural, event, and temporal properties to be proved. Our method, related to finite state automata and using marking techniques, generalises what we did within previous work, where the regular expressions we processed were more restricted. But we can only deal with a subset of first-order logic formulas.

Keywords Component-based approach, dynamic reconfiguration paths, multiple reconfiguration paths, checking invariance properties, finite state automata, marking techniques.

1 Introduction

Dynamic reconfigurations of software architectures are active research topics [1, 3, 5, 19, 20, 21, 18, 25]. They provide large increase in value for component-based software. Such an approach allows some components to be replaced or removed, in particular if they fail. In order to provide more services, more components may be added dynamically, too. So dynamic reconfigurations increase the availability and reliability of such systems by allowing their architecture to evolve at run-time.

The work presented hereafter is an extension of [14], which addresses the verification of architectural, event, or temporal properties. Such properties may be crucial for systems with high-safety requirements. About the definition of such properties, [9] proposes FTPL1, a temporal logic for dynamic reconfigurations applied to components defined by means of the Fractal toolbox [4] and including such properties. FTPL allows successive reconfigurations—modelled by reconfiguration paths—to be applied to successive configurations (or component models). Since FTPL is based on first-order predicate logic, such properties are undecidable in general, there only exist partial solutions for proving them.

Many authors developed methods that work whilst software is running and may be reconfigured—e.g., [17, 18], based on FTPL, or [12] as another example. Therefore we know if a property holds for the successive members of a chain of reconfigurations, until the current run-time state. Our method is very different, 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. In fact, we do not verify such properties at run-time, 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. Of course, we cannot consider reconfigurations caused by totally unexpected events but we think that our approach is complementary to such works, our goal is to go as far as possible within this static approach. In [14], we proposed a method based on this point of view and using marking techniques related to model-checking:

1Fractal Temporal Pattern Logic.
given a reconfiguration path that may be applied when the software is running, we aimed to ensure that a property holds if this path is actually applied when the software works. We were able to deal with some cases of infinite reconfiguration paths, but we only processed one possible reconfiguration path. Dealing with only one path is not restrictive for methods applied at run-time, whilst the software is working, but is rather limited at design-time, where several possible futures could be studied. In the present article, we propose the new notion of multiple reconfiguration paths, which are expressions denoting several possible reconfiguration processes. However, this extension has a price: the correctness of our new implementations—w.r.t. the definitions of [9]—is guaranteed only for a strict subset of formulas, in comparison with formulas used within [14].

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, 17]. Section 3 precisely introduces our notion of multiple reconfiguration path and Section 4 recalls the organisation of our framework. Then we give updated versions of our programs in Section 5 and study the correctness of these implementations w.r.t. the operators defined in Section 2. We do not examine all the operators, but our examples are representative: implementation techniques and correctness proofs are analogous. Section 6 discusses some advantages and drawbacks of our method, in comparison with other approaches. It also introduces future work. In order for this article to be self-contained, most of the definitions put hereafter are identical to [14]’s. Readers familiar with that article can skip Section 2—except for the definition of the CP♭ set—and § 4.1.

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, 17] 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 $\mathcal{S}$ be a set of type names $^2$ a component $\mathcal{C}$ is defined by:

- three pairwise-disjoint sets of parameters $^3$ $P_\mathcal{C}$, input port names $I_\mathcal{C}$, and output port names $O_\mathcal{C}$;
- the class $t_\mathcal{C}$ encompassing the services implemented by the component;
- additional functions to get access to the class of a parameter or port ($\tau_\mathcal{C} : P_\mathcal{C} \cup I_\mathcal{C} \cup O_\mathcal{C} \rightarrow \mathcal{S}$, or to a parameter’s value ($v_\mathcal{C} : P_\mathcal{C} \rightarrow \bigcup_{s \in \mathcal{S}} s$);
- the set sub-$\mathcal{C}_\mathcal{C}$ of its subcomponents if the $\mathcal{C}$ component is composite$^4$;
- the set $B$ of bindings of ports—that is, couples of input and input port names, being the same type, and the set $D$ of delegation links, between composite component ports and port of contained components.

---

$^2$... or class names within an object-oriented approach.

$^3$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.

$^4$Of course, the binary relation ‘is a subcomponent of’ must be a direct acyclic graph. A composite component cannot have parameters. More precisely, it implicitly has the parameters of all its sub-components.
Possible components of an HTTP server are given in Fig. 1 as an example of a component-based architecture, already used in [6]. Requests are read by the RequestReceiver component and transmitted to the RequestHandler component. When the latter processes a request, it may consult the cache by means of the CacheHandler component or transmit this request to the RequestDispatcher component, which manages file servers. This architecture is based on a cache and load balancer, in order for response times to be as short as possible. The cache must be used only if the number of similar requests is very high, and the amount of memory devoted to the cache component must be automatically adjusted to the Web server’s load. The validity duration of the data put in the cache must also be adjusted with respect to the Web server’s load. In addition, more data servers have to be deployed if the servers’ average load is high. According to these conventions, we see that some components may be added or removed, depending on some parameters.

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

\[ B \ni (\text{cache}_{\text{CacheHandler}}, \text{getCache}_{\text{RequestHandler}}) \]

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 (\(\land, \lor, \ldots\)) and quantifiers (\(\forall, \exists\)). These configuration properties form a set denoted by \(CP\). The subset \(CP^\flat\) is build analogously, but connectors and quantifiers are restricted to \(\land\) and \(\forall\). Roughly speaking, formulas belonging to \(CP^\flat\) are comparable to premises of Horn clauses within logic programming.

2.3 Reconfiguration Operations

Primitive reconfiguration operations apply to a component architecture, and the output is a component architecture, too. 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

\[^5\text{They may be viewed as graph transformations applied to component models if we consider such models as graphs.}\]
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. 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.

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 $\mathcal{R}$. The set of evolution operations is $\mathcal{R}_{\text{run}} = \mathcal{R} \cup \{ \text{run} \}$ where run is an action modelling that all the stopped components are restarted and the software is running.

**Definition 2** ([10, 17]) The operational semantics of component systems with reconfigurations is defined by the labelled transition system $\mathcal{S} = (C, C^0, \mathcal{R}_{\text{run}}, \to, l)$ where $C = \{c, c_1, c_2, \ldots \}$ is a set of configurations—or component models—$C^0 \subseteq C$ is a set of initial configurations, $\mathcal{R}_{\text{run}}$ is a finite set of evolution operations, $\to \subseteq C \times \mathcal{R}_{\text{run}} \times C$ is the reconfiguration relation, and $l : C \to \text{CP}$ is a total function to label each $c \in C$ with the largest conjunction of $cp \in \text{CP}$ evaluated to ‘true’ over $\mathcal{R}_{\text{run}}$.

Let us note $c \xrightarrow{\text{op}} c'$ when a target configuration $c'$ is reached from a configuration $c$ by an evolution $\text{op} \in \mathcal{R}_{\text{run}}$. Given the model $S = (C, C^0, \mathcal{R}_{\text{run}}, \to, l)$, an evolution path $\sigma$ of $S$ is a (possibly infinite) sequence of component models $c_0, c_1, c_2, \ldots$ such that $\forall i \in \mathbb{N}, \exists \text{op} \in \mathcal{R}_{\text{run}}, c_i \xrightarrow{\text{op}} c_{i+1} \in \to$. We write ‘$\sigma[i]$’ to denote the $i$th element of a path $\sigma$, if this element exists. The notation ‘$\sigma[i]$’ denotes the suffix path $\sigma[i], \sigma[i+1], \ldots$ and ‘$\sigma[j]$’ ($j \in \mathbb{N}$) denotes the segment path $\sigma[i], \sigma[i+1], \ldots, \sigma[j-1], \sigma[j]$. An example of evolution 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 deals with 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

---

6The reason: the name of a component—part of its definition—can only identify one component. But you can clone a component under a new name.

7Strictly speaking, we have to stop a component before removing it, and to start it before having added it, as abovementioned. This convention about the run action allows us not to be worried about such stop and start operations within our reconfiguration paths.
Checking Properties along Multiple Reconfiguration Paths for Component-Based Systems

FTPL, in particular those used in the implementations we describe. For more details about this temporal logic, see [10][17]. FTPL’s syntax is defined by:

\[
\begin{align*}
\langle \text{temp} \rangle & ::= \text{after} \langle \text{event} \rangle \langle \text{temp} \rangle \mid \text{before} \langle \text{event} \rangle \langle \text{trace} \rangle \mid \ldots \\
\langle \text{trace} \rangle & ::= \text{always} \ cp \mid \text{eventually} \ cp \mid \ldots \\
\langle \text{event} \rangle & ::= \ op \ \text{normal} \mid \ op \ \text{exceptional} \mid \ op \ \text{terminates}
\end{align*}
\]

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 \( \sigma \) be an evolution path, the FTPL semantics is defined by induction on the form of the formulas as follows[8]—in the following, \( i \in \mathbb{N} \)—:

- for the events:
  \[
  \begin{align*}
  \sigma[i] \models op \ \text{normal} & \quad \text{if} \quad i > 0 \land \sigma[i-1] \neq \sigma[i] \land \sigma[i-1] \overset{op}{\rightarrow} \sigma[i] \in \to \\
  \sigma[i] \models op \ \text{exceptional} & \quad \text{if} \quad i > 0 \land \sigma[i-1] = \sigma[i] \land \sigma[i-1] \overset{op}{\rightarrow} \sigma[i] \in \to \\
  \sigma[i] \models op \ \text{terminates} & \quad \text{if} \quad \sigma[i] \models op \ \text{normal} \lor \sigma[i] \models op \ \text{exceptional}
  \end{align*}
  \]

- for the trace properties:
  \[
  \begin{align*}
  \sigma \models \text{always} \ cp & \quad \text{if} \quad \forall i : i \geq 0 \Rightarrow \sigma[i] \models cp \\
  \sigma \models \text{eventually} \ cp & \quad \text{if} \quad \exists i : i \geq 0 \Rightarrow \sigma[i] \models cp
  \end{align*}
  \]

- for the temporal properties:
  \[
  \begin{align*}
  \sigma \models \text{after} \ event \ temp & \quad \text{if} \quad \forall i : i \geq 0 \land \sigma[i] \models \text{event} \Rightarrow \sigma[i] \models \text{temp} \\
  \sigma \models \text{before} \ event \ trace & \quad \text{if} \quad \forall i : i > 0 \land \sigma[i] \models \text{event} \Rightarrow \sigma[i-1] \models \text{trace}
  \end{align*}
  \]

**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 7:

\[
\text{after AddCacheHandler normal always CacheConnected}
\]

**Remark 5** About temporal and trace properties, let us notice that if such a property holds on an evolution path, it holds on any prefix of this path.

3 Multiple Reconfiguration Paths

**Definition 6** Let \( R_{\text{run}} \) be a set of evolution operations, a reconfiguration path is a sequence of elements of \( R_{\text{run}} \), and the set \( \Omega_{R_{\text{run}}} \) of multiple reconfiguration paths on \( R_{\text{run}} \) is the set of regular expressions built over the alphabet \( R_{\text{run}} \). Let us recall that the constructs used within regular expressions are ‘\(|\)’ for alternatives, ‘\( ?\)’ for an optional occurrence of an alphabet’s member, ‘\( *\)’ (resp. ‘\(+\)’) for zero (resp. one) or more occurrences of such a member. Semantically, a multiple reconfiguration path is the set of all the prefixes of all the reconfiguration paths denoted by this regular expression.

**Example 7** The following multiple reconfiguration path:

\[\text{Example 7} \]

---

[8]: For a complete definition including all the operators, see [10].
run RemoveCacheHandler AddCacheHandler
(\text{MemorySizeUp} \ run
(\text{AddFileServer} \ \text{DurationValidityUp} \ | \ \text{DurationValidityUp} \ \text{AddFileServer}) \ \ run?
\text{DeleteFileServer})^+ \ \text{AddFileServer}

includes the chain of reconfigurations pictured at Fig. 2.

Remark 8 Let us recall that a reconfiguration path may be infinite. Looking at Ex. 7, we consider that the ‘(...)\text{+}’ expression can be iterated a finite number of times, followed by the \text{AddFileServer} operation; another possible behaviour is an endless iteration of the ‘(...)\text{+}’ expression. We encompass all these possible behaviours by considering prefixes, as mentioned in Def. 6.

It is well-known for many years—since Kleene’s theorem—that a regular expression language can be recognised by a deterministic finite state automaton, whose transitions are labelled by members of this language’s alphabet. Let us recall that such an automaton \mathcal{A} is defined by a set \( Q \) of \textit{states}, a set \( L \) of \textit{transition labels}, and a set \( T \subseteq Q \times L \times Q \) of \textit{transitions}. As in Def. 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. As an example, Ex. 7’s language can be recognised by the automaton pictured in Fig. 3 (the states \( q_0, q_1, q'_1, \ldots, q_5 \) have been respectively named in connection to the successive component models \( c_0, c_1, c'_1, \ldots, c_5 \) of Fig. 2). In addition, let us recall that such an automaton can be build automatically from a regular expression. In the next section, we explain what our states are, and which operations are performed by our transitions.

4 Our Method’s Bases

4.1 Modus Operandi

As mentioned above, our framework’s basis is an automaton modelling the possible evolution paths of a multiple reconfiguration path. A state of such an automaton 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. Such an automaton has
an initial state, given by the initial component model \((q_0\text{ in Fig. 3})\). Since we aim to recognise all the prefixes of possible reconfiguration paths, any state may be viewed as final. In addition, since some infinite behaviours are accepted (e.g., endlessly cycling from the \(q_5\) or \(q_6\) state to the \(q_2\) state in Fig. 3), there are processes without ‘actual’ final state. In fact, the complete automaton may be viewed as an \(\omega\)-automaton. Let us go back to states reached several times—e.g., the \(q_2\) state in Fig. 3 reached after \(q_5\) and \(q_6\)—: considering that the whole system is back to a previous state may be not exact, because some parameters may 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 beginning of § 2.2.

Several programming languages are used within our framework. 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\(^9\) we use for our component models is TACOS+/XML \(^13\). This language using XML\(^10\)-like syntax is comparable with other ADLS, in particular 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, a short example is given in \(^14\). Reconfigurations operations are implemented using XSLT\(^11\): the input and output are TACOS+/XML files.

When we model that 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. In the programs given below, we compute each component model belonging to a reconfiguration path. For each component model, we may verify topological properties, e.g., checking that a component or binding is present. As in \(^14\), these topological properties are computed by means of XQuery programs \(^27\). There is no difficulty about the implementation of reconfiguration operations and property checks, so the descriptions put hereafter concern the part implemented by means of automata.

### 4.2 Types Used

Now we describe our checking functions at a high level. First we make precise the types used, in order to ease the reading of our functions. The formalism we use is close to type definitions in strong typed functional programming languages like Standard ML \(^23\) or Haskell \(^22\). Of course, we assume that some types used hereafter—e.g., ‘bool’, ‘int’—are predefined. We use the same names than in \(^14\) for identical notions, and new functions introduced are suffixed by ‘*’ or ‘**’.

As mentioned above, 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

\(^{9}\text{Architecture Definition Language.}\)

\(^{10}\text{eXtensible Markup Language.}\)

\(^{11}\text{eXtensible Stylesheet Language Transformations, the language of transformations used for XML documents.}\)

\(^{12}\)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.
introduce these two function types as:

\[
\text{type } \text{evolution-op} = \text{component-model} \rightarrow \text{component-model} \\
\text{type } \text{check-property} = \text{component-model} \rightarrow \text{bool}
\]

An event is defined by an evolution operation and a symbol related to this operation’s result (cf. Def. 3):

\[
\begin{align*}
\text{function } \text{event} \rightarrow \text{ev-op} & : \text{event} \rightarrow \text{evolution-op} \\
\text{function } \text{event} \rightarrow \text{termination-s} & : \text{event} \rightarrow \text{termination-symbol} \\
\text{type } \text{termination-symbol} & = \{ \text{normal, exceptional, terminates} \}
\end{align*}
\]

This last information is used by a function checking that the component model got by an evolution operation and the previous component model are equal or different, depending on this symbol\textsuperscript{12}:

\[
\text{function } \text{term-check} : \text{event} \rightarrow (\text{component-model} \times \text{component-model} \rightarrow \text{bool})
\]

Let state be the type used for a state of our automata, starting from such a state and a configuration\textsuperscript{13} is expressed by the following type:

\[
\text{type } \text{path-check} = \text{state} \times \text{component-model} \rightarrow \text{bool}
\]

The following function yields all the transitions starting from a state:

\[
\text{function } t : \text{state} \rightarrow \text{set-of}[\text{transition}]
\]

the data belonging to a set can be accessed by means of a ‘for’ expression. A transition starts from a state and returns a state, and the label of such a transition is given by the l function:

\[
\begin{align*}
\text{type } \text{transition} & = \text{state} \rightarrow \text{state} \\
\text{function } l & : \text{transition} \rightarrow \text{evolution-op}
\end{align*}
\]

In the following, we will focus on the constructs ‘after’ and ‘always’. The path-check type is used within:

\[
\begin{align*}
\text{function } \text{check-after*} & : \text{evolution-op} \times \text{path-check} \rightarrow \text{path-check} \\
\text{function } \text{check-always*} & : \text{check-property} \rightarrow \text{path-check}
\end{align*}
\]

In other words, check-always*(check-p*)(q,c) applies the check-p* function along the q state, the states reached by transitions originating from q, 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 functions such as check-before* and check-eventually* (cf. § 2.4).

\textsuperscript{12}Let us recall (cf. Def. 3) that if this symbol is ‘terminates’, no additional checking is performed.

\textsuperscript{13}That is, a component model (see Def. 2).
4.3 Ordering States of Automata

In this section, we introduce some notions related to our automata and used in the following. The states of our automata modelling multiple reconfiguration paths can be ordered with respect to the transitions performed before cycling. Let $\mathcal{A}$ be an automaton, $q_0$ its initial state, $L$ its set of transition labels, and $T$ its set of transitions, if $q$ and $q'$ are two states of $\mathcal{A}$:

\[
q \rightarrow q' \overset{\text{def}}{\iff} \exists \tau \in T, \exists l \in L, \tau = (q, l, q') \quad \text{[By language abuse, we note } q' = \tau(q).]
\]

\[
q < q' \overset{\text{def}}{\iff} q = q_0 \lor \{ (q_1, \ldots, q_n, q_1', \ldots, q_p') \mid q_0 \rightarrow q_1 \rightarrow \cdots \rightarrow q_n \rightarrow q \rightarrow q_1' \rightarrow \cdots \rightarrow q_p' \rightarrow q' \}
\]

and $q_0, q_1, \ldots, q_n, q, q_1', \ldots, q_p', q'$ are pairwise-different. The notation '$q \leq q'$ stands for '$q < q' \lor q = q'$. If we consider the $\mathcal{A}_0$ automaton pictured at Fig. 3, $q_0 < q_1 < q_1' < q_2 < q_3 < q_3' < q_4 < q_5 < q_6 < q_8$ and $q_3' < q_7 < q_5$. Obviously, our '$<$' relation is a partial order.

**Remark 9** In fact, we build a binary relation step by step by exploring all the possible paths from the initial state, until we reach a state previously explored within the same chain, and our '$<$' function is the transitive closure of this relation. As a consequence, the transitions which do not satisfy this property are those going back to a state already explored.

5 Our Method’s Functions

5.1 Our Markers

Our main idea—already expressed in [14]—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 states of a multiple reconfiguration path’s automata. 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;
- **checked**: the property has already been checked, and no additional check is needed if this step is explored again.

However, there is a significant difference between [14] and the present work: in [14], one marker was used for a state. This is impossible here since we have to explore several possible transitions from a same state. Let us consider the multiple reconfiguration path $((e \mid op_0) op_1)^+$—where $e, op_0, op_1 \in R_{\text{run}}$ with $op_0 \neq e, op_1 \neq e$—and a property after $e$ always $cp$. When this regular expression is resumed, there are two cases: either the $e$ event has been recognised, in which case we have to check the $cp$ property on all the successive states and cycling is detected after the new application of the $e$ operation, or $op_0$ and $op_1$ have been performed and we are still waiting for the $e$ event. We cannot use the same markers for these two cases.

The type of the check-after* function is given in § 4.2. In fact, an automaton modelling a multiple reconfiguration path is pre-processed and its states are marked as unchecked, by means of a new mark, mark-for-after. Then a recursive function check-after**—being the same type—is launched, reads and updates this new mark. The check-always* function behaves the same, the recursive function which is launched is check-always** and the new marker is mark-for-always.
The implementation of the functions check-after** and check-always** is given in Fig. 5. We use a high-level functional pseudo-language, except for updating marks, which is done by means of side effects. A more complete implementation is available at [15], 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].

5.2 Implementations’ Correctness

Concerning the termination of the functions check-after** and check-always**, the proofs are similar to those given in [14]. The correctness is also ensured for idempotent reconfiguration operations, excluding some operations on parameters, but proofs are here more subtle.

5.2.1 Termination

Proposition 10 The function check-after** terminates.

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

$$\text{check-after**}(e, \text{check-f}*)(q_0, c)$$

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 $\forall q_j : q_0 \leq q_j < q_i, \text{mark-for-after}(q_j) = \text{again}$ when it is applied to the $q_i$ state. The transitions which may be fired from $q_i$ are a finite set, so the ‘for’ loop terminates if for each transition, the process terminates. Let $q_k$ be a state reached from $q_i$. If $q_i < q_k$, the
invariant holds. If \( q_i \not< q_k \), then \( q_k \) is a state already explored\(^{[14]}\) that is, the next recursive call applies to a state whose the value of mark-for-after is again. Such a call terminates.

**Proposition 11** The function check-always** terminates.

This termination proof is similar: since transitions which may be fired from \( q_i \) are a finite set, the 'for' loop terminates if for each transition, the process terminates. However, let us notice that a process launched by the check-always** function 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)), where \( e \) is a reconfiguration operation and \( cp \) a formula. Before reaching the end of a cycle, the invariant is:

\[
\forall q_j : q_0 \leq q_j < q_i, \text{mark-for-always}(q_j) = \text{checked} \lor \text{mark-for-always}(q_j) = \text{again}
\]

when the check-always** function is applied to the \( q_i \) state. Roughly speaking, when a cycle is performed, this 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, at a \( q_0 \) state already explored, the invariant is \( \forall q_j : q_0 \leq q_j < q_i, \text{mark-for-always}(q_j) = \text{checked}, q_i \) being the current state. Let \( q_k \) a state reached from \( q_i \). If \( q_i < q_k \), the invariant holds. If \( q_i = q_0 \), this recursive call of check-always** is performed with the situation:

\[
\forall q_j : q'_0 \leq q_j < q'_0, \text{mark-for-always}(q_j) = \text{checked}
\]

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

**5.2.2 Restrictions on Formulas**

Let us recall that in \([14]\), we were able to deal with finite paths and cycles without continuation, that is, the ‘+’ construct of regular expressions was used only at a final position. In other words, there were no alternatives. In this previous work, we also mentioned that our modus operandi is suitable if the cycle of reconfiguration operations is idempotent. Since the composition of two commutative idempotent functions is idempotent, too, some pairs of reconfiguration operations can be commuted, some consists of operations which neutralised each other, and globally, most cycles used are globally idempotent. Concerning our primitive reconfigurations, most of them are idempotent, e.g., a component’s addition or removal, as well as a binding’s addition or removal. Assigning a constant value to a parameter is idempotent, but general changes are not, e.g., incrementing or decrementing a parameter.

Of course, this limitation still holds for our revised algorithms. Another limitation exists for alternative with a common continuation. As a simple counter-example, let us consider the multiple reconfiguration path \((op_0 \mid op_1) op_2\). If we process the formula always\( cp—cp \in CP\)—our algorithm checks the \( cp \) formula at the initial state, then at the result of \( op_0 \), then at the result of \( op_2 \) after \( op_0 \). The result of \( op_1 \) applied to the initial state is checked, and the process stops because of the mark put at the common state after \( op_0 \) and \( op_1 \). Now let \( cp \) be \( cp_0 \lor cp_1 \)—where \( cp_0, cp_1 \in CP \)—and let us assume that \( cp_0 \land \neg cp_1 \) (resp. \( \neg cp_0 \land cp_1 \)) holds on the result of \( op_0 \) (resp. \( op_1 \)). If \( cp_1 \) is always false after applying \( op_2 \)—e.g., \( cp_1 \) may be related to a binding removed by \( op_2 \)—our method results in an erroneous answer along the path \( op_1 op_2 \), even it is right for the path \( op_0 op_2 \).

\(^{[14]}\)See Rem.\(^{[9]}\)
Solutions exist. We could restrict alternatives of regular expressions by allowing them only at the top level. The counter-example above would be rewritten as \((op_0 \ op_2 \ | \ op_1 \ op_2)\), the result of \(op_2\)—as a component model—would be checked twice, one time after applying \(op_0\), the second after applying \(op_1\). Adopting such a rule would complicate the processing of a multiple reconfiguration path such as \((op_0 \ | \ op_1)^+\). Another drawback is that a multiple reconfiguration path may contain alternatives for the corresponding automaton even if the ‘|’ operator is not used explicitly. As an example, let us consider the multiple reconfiguration path \(op_0 \ op_1? \ op_2\). The alternative syntactically appears if we rewrite it by means of a grammar—\(S\) being the axiom, \(S'\) another non-terminal symbol, and \(\varepsilon\) the empty word—:

\[
S \rightarrow op_0 \ S' \ op_2 \\
S' \rightarrow op_1 \ | \ \varepsilon
\]

and an analogous counter-example, based on a logical disjunction, can be found for such a case. This drawback does not appear if a non-empty cycle is possibly followed by a continuation, that is, in a multiple reconfiguration path like \(op_0^+ \ op_1\). If we rewrite this example by means of a grammar:

\[
S \rightarrow op_0 \ S' \\
S' \rightarrow op_0 \ S' \ | \ op_1
\]

we will see that no common part follows the alternative. This is different if the cycle can be empty. As an example, the multiple reconfiguration path \(op_0 \ op_1^* \ op_2\) can be rewritten using the following grammar:

\[
S \rightarrow op_0 \ S' \ op_2 \\
S' \rightarrow op_1 \ S' \ | \ \varepsilon
\]

and a common part follows the alternative.

From our point of view, the best solution is to restrict formulas to the strict subset \(CP^\flat\) defined in § 2.2. In other words, the ‘\(\lor\)’ connector must not be used, the ‘\(\forall\)’ quantifier—related to that connector—and the ‘\(\neg\)’ operator must not, either.

5.2.3 Correctness for Restricted Formulas

Adopting these additional conventions, proving the correctness of our function \texttt{check-always*}—other functions’ correctness is analogous—is tedious but not really difficult. We have to examine all the basic cases of formulas \(cp \in CP^\flat\)—e.g., the set membership of a binding—and idempotent reconfiguration operations \(op_0, op_1, op_2\) to show the following proposition.

\textbf{Proposition 12} Starting from the same state and the same component model, if the formula \texttt{always cp}—where \(cp \in CP\)—holds on the two paths \(op_0 \ op_2\) and \(op_1\)—that is, before and after applying \(op_1\)—it also holds on the multiple reconfiguration path \((op_0 \ | \ op_1) \ op_2\).

By induction, it is easy to prove such a property about longer paths. It is also easy to prove that if this property holds for the two formulas \(cp_0\) and \(cp_1\), it also holds for the formula \(cp_0 \ \land \ cp_1\). An analogous proof exists for the ‘\(\lor\)’ quantifier. By induction on the number of members of a multiple reconfiguration path, we can prove this proposition by considering a grammar associated with this path, as we sketch in § 5.2.2. As a consequence, if a same state is reached along several paths, the property holds and our function \texttt{check-always**} is correct. Studying the correctness of the function \texttt{check-after**} is easier, because the possible futures of each path of an alternative are explored independently.

6 Discussion and Future Work

Within the framework sketched at § 4.1, the new versions of our programs have been implemented using the Java programming language and can be found in [15]. 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
marking techniques and used in model-checking, e.g., [7, 8, 24].

As mentioned in the introduction, our method takes place at design-time. We do not deal with a
language to describe reconfiguration operations and constraints on these operations as an extension of
an ADL, as in [25], we are mainly interested in developing effective methods for verifying properties.
In [14] we were able to deal with a particular case of infinite paths, based on the fact that often the
same sequences are repeated: a component may be stopped in some circumstances, restarted in some
circumstances, and so on. However, it is true that this situation was restrictive and the initial motivation
of the present work was to introduce alternatives within our paths. Such construct would be irrelevant
within methods working at run-time [17, 18], since they observe a process in progress, the history of
reconfiguration operations being known. At design-time, it may be interesting to plan several possible
behaviours, what is new in comparison with [14]. In the present work, we choose to focus on some
efficiency for our algorithms, since common parts are explored once and cycles are explored two times
at most, that is, our algorithms are linear with respect to the automaton’s state number. In other words,
we are able to explore several possible behaviours quite efficiently, but the price to pay is a restriction
of the formulas processed. However, if we look at the examples given within [9, 10, 11, 17], we can think
that our restriction is not too cumbersome in practice.

As mentioned above, other solutions exist, but we wanted our extension to be close to our original
modus operandi. If we consider a ‘simple’ reconfiguration path, that is, only one transition starts
from each state of the corresponding automaton, we get exactly the programs given in [14]. Yet an-
other work may consider only alternatives without syntactic common continuation—possibly by applying
some transformation rules—or our algorithms could be changed in order to explore more states in
such a case, but this second solution might lead to some combinatorial explosion. Another solution could
be based on branching-time logic for reconfiguration alternatives, whereas the present work is based on
linear-time logic, as in [9, 10, 11, 17]. Other ideas could be based on a connection with the Model Driven
Engineering technical space [2], who would provide more expressive power. Likewise, we could plan a
bridge between our approach and others, closer to a semantic level: for example, [19] models reconfig-
uration operations by means of graph rewriting and uses formal verification techniques along graphs to
check properties related to reconfigurations.

On another point, we are interested in this work in reconfigurations, but not in reasons for these re-
configurations [15], most often expressed by reconfiguration policies [6]. In parallel, we are working on an
extension of [14] taking such policies into account [16]. In the future, we plan to integrate reconfiguration
policies into our approach based on multiple reconfiguration paths.

7 Conclusion

In comparison with methods at run-time, ours may appear as too static, unable to cope with unexpected
situations. Our plan is to investigate as far as possible properties that can be checked at design-time,
in order for a reconfigurable system to be deployed as safely as possible. Our work can be used for
simulations, it may help conceptors design policies involving reconfigurations with good properties. Our
tool is not ready for testing policies, but can be used for testing possible results of policies. We see that
such an approach does not aim to replace works applied at run-time, but to complement them. About
examples such as an HTTP server, we succeeded in proving properties. In other words, we think that our
method can provide some significant help at design-time.

15This is the same in [14].
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 pointed out some omissions and suggested me constructive improvement.

References


