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 FTPL[1], 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:

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

J. Kofroň, J. Tumova (Eds.): Formal Engineering approaches
to Software Components and Architectures (FESCA'17)
EPTCS 245, 2017, pp. 52–66, doi:10.4204/EPTCS.245.5

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 $CP^\flat$ 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 $\mathscr{S}$ be a set of *type names* [2], a *component* $\mathscr{C}$ is defined by:

- three pairwise-disjoint sets of *parameters*[3] $P_\mathscr{C}$, *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 ($v_\mathscr{C} : P_\mathscr{C} \to \bigcup_{s \in \mathscr{S}} s$);
- the set *sub*-$c_\mathscr{C}$ of its subcomponents if the $\mathscr{C}$ 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.

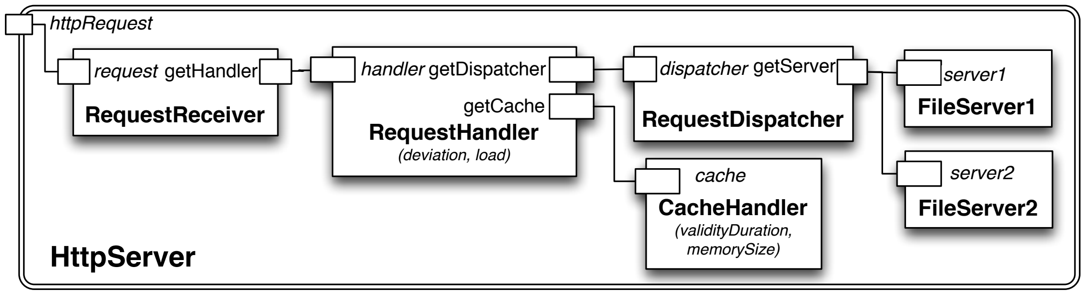

Figure 1: Component-based architecture of an HTTP server [11].

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:*

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

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. The subset $CP^\flat$ is build analogously, but connectors and quantifiers are restricted to $\wedge$ 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[5]. 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]They may be viewed as *graph transformations* applied to component models if we consider such models as graphs.

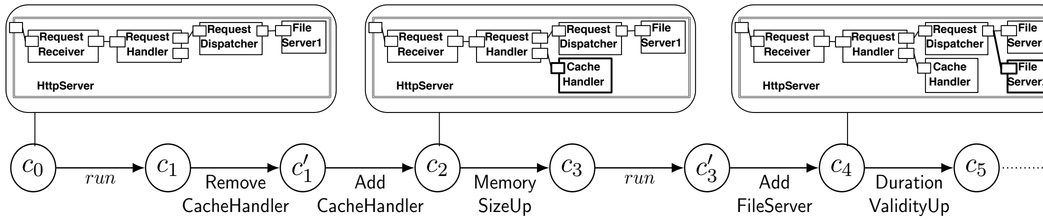

Figure 2: Part of an evolution path of Fig. 1's HTTP server architecture [11].

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[6]. 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[7].

Definition 2 ([10, 17]) *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—or component models—*$C^0 \subseteq C$ *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* $\mathscr{R}_{run}$.

Let us note $c \stackrel{op}{\rightarrow} c'$ 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 $\forall i \in \mathbb{N}, \exists op \in \mathscr{R}_{run}, c_i \stackrel{op}{\rightarrow} c_{i+1} \in \rightarrow$. We write '$\sigma[i]$' to denote the ith element of a path σ, if this element exists. 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 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

[6]The 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.

[7]Strictly 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.

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:

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

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*[8]*—in the following, $i \in \mathbb{N}$—:*

- for the events:

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

- for the trace properties:

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

- for the temporal properties:

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

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`

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 $\mathscr{R}_{run}$ be a set of* evolution operations*, a* **reconfiguration path** *is a sequence of elements of $\mathscr{R}_{run}$, and the set $\Omega_{\mathscr{R}_{run}}$ of* **multiple** *reconfiguration paths on $\mathscr{R}_{run}$ is the set of* regular expressions *built over the alphabet $\mathscr{R}_{run}$. 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:*

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

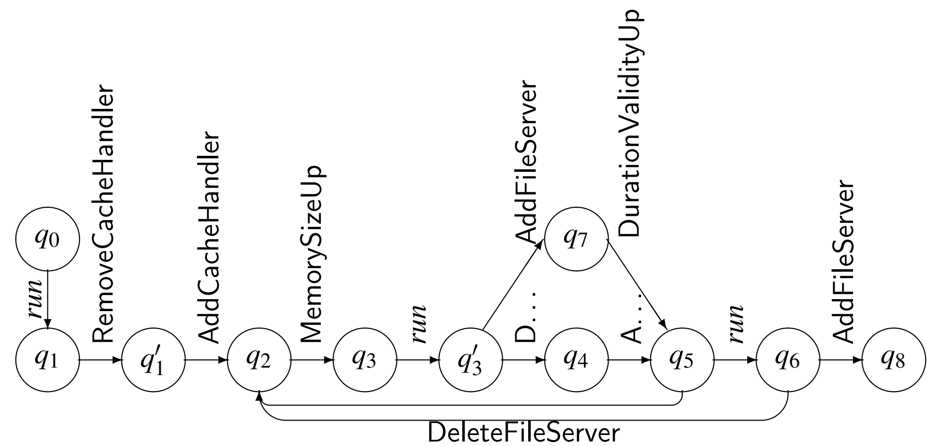

Figure 3: Automaton for a multiple reconfiguration path.

run RemoveCacheHandler AddCacheHandler
*(*MemorySizeUp *run*
(AddFileServer DurationValidityUp | DurationValidityUp AddFileServer*)* *run?*
DeleteFileServer*)+* 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 '*`(...)+`*' expression can be iterated a finite number of times, followed by the* AddFileServer *operation; another possible behaviour is an endless iteration of the '*`(...)+`*' 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 $\mathscr{A}$ 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*. As in Def. 3 for systems with reconfigurations, there exists a function $l : Q \to CP$, 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 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 ω-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.

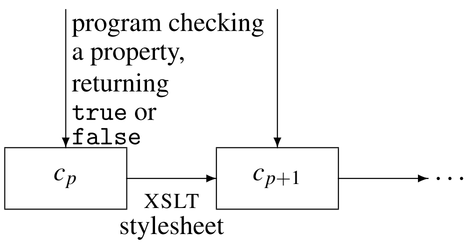

Figure 4: Our organisation.

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

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

[11] e**X**tensible **S**tylesheet **L**anguage **T**ransformations, the language of transformations used for XML documents [26]. 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:

type `evolution-op   = component-model` $\to$ `component-model`
type `check-property = component-model` $\to$ `bool`

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

function `event->ev-op          :  event` $\to$ `evolution-op`
function `event->termination-s :  event` $\to$ `termination-symbol`
type `termination-symbol       =  {normal,exceptional,terminates}`

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[12]:

function `term-check :  event` $\to$ `(component-model` $\times$ `component-model` $\to$ `bool)`

Let `state` be the type used for a state of our automata, starting from such a state and a configuration[13] is expressed by the following type:

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

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

function `t :  state` $\to$ `set-of[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:

type `transition = state` $\to$ `state`
function `l :  transition` $\to$ `evolution-op`

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

function `check-after*  :  evolution-op` $\times$ `path-check` $\to$ `path-check`
function `check-always* :               check-property` $\to$ `path-check`

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).

[12] Let us recall (cf. Def. 3) that if this symbol is '`terminates`', no additional checking is performed.
[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 $\mathscr{A}$ 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 $\mathscr{A}$:

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

$$q < q' \overset{\text{def}}{\iff} q = q_0 \vee \begin{cases} \exists (q_1, \ldots, q_n, q'_1, \ldots, q'_p), \\ q_0 \mapsto q_1 \mapsto \cdots \mapsto q_n \mapsto q \mapsto q'_1 \mapsto \cdots \mapsto q'_p \mapsto q' \end{cases}$$

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 '$q < q' \vee q = q'$'. If we consider the $\mathscr{A}_0$ automaton pictured at Fig. 3, $q_0 < q_1 < q'_1 < q_2 < q_3 < q'_3 < q_4 < q_5 < q_6 < q_8$ 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 $((\mathtt{e} \mid op_0)\ op_1)+$—where $\mathtt{e}, op_0, op_1 \in \mathscr{R}_{run}$ with $op_0 \neq \mathtt{e}$, $op_1 \neq \mathtt{e}$—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`.

```
check-after**(e,check-f*)(q,c) ⟶
  if mark-for-after(q) == again then true
  else     // mark-for-after(q) == unchecked
    mark-for-after(q) ⟵ again ; result ⟶ true ;
    for τ in t(q) do
      c0 ⟵ l(τ)(c) ; q0 ⟵ τ(q) ;
      result ⟶ result and
                 if l(τ) == event->ev-op(e) and event->termination-s(e)(c0,c) then check-f*(q0,c0)
                   else check-after**(e,check-f*)(q0,c0)
                 end if
    end for ;
    result ;
  end if
end

check-always**(check-p*)(q,c) ⟶
  check-p*(c) ∧ if mark-for-always(q) == checked then true
               else     // mark-for-always(q) ∈ {unchecked,again}
                 mark-for-always(q) ⟵ checked ; result ⟵ true ;
                 for τ in t(q) do
                   c0 ⟵ l(τ)(c) ; q0 ⟵ τ(q) ; result ⟶ result and check-always**(check-p*)(q0,c0)
                 end for ;
                 result ;
               end if ;
end
```

Figure 5: Checking properties: two implementations.

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:

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

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 $\forall q_j : q_0 \leq q_j < q_i, \texttt{mark-for-after}(q_j) = \texttt{again}$ 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 $\texttt{check-after*}(e, \texttt{check-always*}(cp))$, where e is a reconfiguration operation and cp a formula. Before reaching the end of a cycle, the invariant is:

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

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 $\forall q_j : q_0' \leq q_j < q_i, \texttt{mark-for-always}(q_j) = \texttt{checked}$, 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:

$$\forall q_j : q_0' \leq q_j \not< q_0', \texttt{mark-for-always}(q_j) = \texttt{checked}$$

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 $cp_0 \vee cp_1$—where $cp_0, cp_1 \in CP$—and let us assume that $cp_0 \wedge \neg cp_1$ (resp. $\neg cp_0 \wedge cp_1$) 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 $(op_0\ op_2 \mid op_1\ op_2)$, 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 $(op_0 \mid op_1)+$. 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 ε the empty word—:

$$\mathtt{S} \longrightarrow op_0\ \mathtt{S'}\ op_2 \qquad\qquad \mathtt{S'} \longrightarrow op_1 \mid \varepsilon$$

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:

$$\mathtt{S} \longrightarrow op_0\ \mathtt{S'} \qquad\qquad \mathtt{S'} \longrightarrow op_0\ \mathtt{S'} \mid op_1$$

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:

$$\mathtt{S} \longrightarrow op_0\ \mathtt{S'}\ op_2 \qquad\qquad \mathtt{S'} \longrightarrow op_1\ \mathtt{S'} \mid \varepsilon$$

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 '$\vee$' 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 `check-always*`—other functions' correctness is analogouss—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.

Proposition 12 *Starting from the same state and the same component model, if the formula* always *cp—where $cp \in CP^{\flat}$—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 $(op_0 \mid op_1)\ op_2$.*

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 $cp_0 \wedge cp_1$. An analogous proof exists for the '$\forall$' 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 `check-always**` is correct. Studying the correctness of the function `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 tehniques 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 another 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 reconfiguration 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 reconfigurations[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 mutiple 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.

[15] This 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

[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] Franck Chauvel, Olivier Barais, Isabelle Borne & Jean-Marc Jézéquel (2008): *Composition of Qualitative Adaptation Policies*. In: *Proc. ASE'08*, L'Aquila, Italy, pp. 455–458, doi:10.1109/ASE.2008.72.

[7] 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.

[8] 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.

[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 (2015): *Using Model-Checking Techniques for Component-Based Systems with Reconfigurations*. In Bara Buhnova, Lucia Happe & Jan Kofroň, editors: *Proc.* FESCA *2015*, EPTCS 178, London, United Kingdom, pp. 33–46, doi:10.4204/EPTCS.178.4.

[15] Jean-Michel Hufflen (2017): *Checking Properties of Component-Based Systems Reconfigured by Means of Adaptation Policies—The Programs*. `http://members.femto-st.fr/jean-michel-hufflen/en/tacos-plus`.

[16] Jean-Michel Hufflen (2017): *Using Model-Checking Techniques for Studying Reconfiguration Policies of Component-Based Systems*. Working paper.

[17] 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.

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

[19] 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.

[20] 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.

[21] 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.

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

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

[24] 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.

[25] Alejandro Sanchez, Alexandre Madeira & Luís S. Barbosa (2015): *On the Verification of Architectural Reconfigurations*. *Computer Languages, Systems & Structures* 44, pp. 218–237, doi:10.1016/j.cl.2015.07.001.

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

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