Testing Conformance in Multi-component Enterprise Application Management

Jacopo Soldani[1(✉)], Lars Luthmann[2], Malte Lochau[2], and Antonio Brogi[1]

[1] University of Pisa, Pisa, Italy
{soldani,brogi}@di.unipi.it
[2] TU Darmstadt, Darmstadt, Germany
{lars.luthmann,malte.lochau}@es.tu-darmstadt.de

Abstract. Modern enterprise applications integrate various heterogeneous components, which management has to be suitably coordinated. Being able to check whether the management allowed by the implementation of an application component conforms to a given specification hence becomes crucial. One may indeed wish to replace component specifications with conforming implementations, by ensuring that already planned management can be enacted, or that no additional (potentially undesired) management activities get enabled. In this perspective, we propose a parametric relation for testing the conformance of the management of application components, based on an existing formalism to model multi-component application management (i.e., management protocols). We also discuss how such relation can be exploited to ensure that replacing a specification with a conforming implementation continues to enable all already allowed management activities, and/or that no additional (potentially undesired) management activity gets enabled.

1 Introduction

Automating the management of enterprise applications is currently a major issue in IT [13]. Enterprise applications indeed integrate various components, and automating the management of an application requires to suitably coordinate the deployment, configuration and operation of its components [8]. This must be done by considering all dependencies and interactions occurring among application components, and the possibility of such components to fail or get stuck [15].

Replaceability is also to be supported [14], as application administrators may wish to replace the specification of desired components with suitable implementations. In this perspective, for suitably replacing a component specification, a candidate implementation must not only implement the specified business logic, but also conform to the specified management. The latter would indeed mean that the implementation of a component can be managed by executing the specified management operations in the specified order, that it properly interacts with the other components forming an application, and that it handles potential failures as specified. In other words, a proper notion of "management conformance"

Published by Springer Nature Switzerland AG 2020
A. Brogi et al. (Eds.): ESOCC 2020, LNCS 12054, pp. 3–18, 2020.
https://doi.org/10.1007/978-3-030-44769-4_1

would hence allow application administrators to replace the specification of a component with implementations that can be managed as specified, by also getting guarantees on the way the implemented component interacts with the rest of the application (and on the overall application management automation).

To this end, we define the notion of management conformance based on *management protocols* [6], an existing approach for modelling multi-component application management. Intuitively, a management protocol specifies the management behaviour of a component by means of a finite state machine, which states model component states, and which transitions indicate which management operations can be performed in a state. States and transitions are enriched with conditions on the requirements of a component, and on the capabilities it offers to satisfy the requirements of other components (bound to such capabilities). Such conditions indicate which requirements must be satisfied while residing in a state or to perform a transition, and the capabilities offered during such state or transition to satisfy the requirements of other components. If some requirements assumed in a state stop being satisfied, a fault handler explicitly specifies how the component should react to such failure.

To define management conformance, we follow Tretmans' idea of input/output (I/O) conformance testing [21], by expressing the semantics of management protocols in terms of I/O labelled transition systems (IOLTS). We then exploit such semantics to define a parametric relation for testing management conformance, which can be instantiated into four different conformance testing relations. We focus on I/O conformance testing rather than on formal verification for two main reasons. I/O conformance testing is known to (i) be more suited for black-box scenarios [21]. It indeed allows us to test whether an existing third-party component conforms to a given specification, even if the such component is a "black-box", with the tester having no clue on how it has been implemented. Conformance testing is also known to (ii) provide a higher degree of implementation freedom, as it delegates to developers the choice of how to implement some under-/non-specified behaviour [19].

We then show how to instantiate the parametric relation for testing management conformance into four different relations. We also discuss how such relations can be used to check the replaceability of the specification of a component with a conforming implementation, as well as how they constitute different trade-offs among implementation freedom and guarantees obtained after replacing a specification with a conforming implementation. The choice of which conformance relation to employ hence depends on the context and requirements of the tester, who can reduce the amount of conforming implementations by considering relations fully preserving already allowed application management, or ensuring that no novel, potentially undesired management activity gets allowed.

To summarise, the contributions of this paper are threefold. We provide (a) an IOLTS semantics for the management protocols modelling the management behaviour of application components. We present (b) a parametric relation for testing management conformance (i.e., testing whether the management protocol of a component implementation conforms to that of a component specification),

which permits instantiating four different conformance testing relations. We discuss (c) whether/how each relation ensures preserving the overall management of an application after replacing a component specification with a conforming implementation, or avoiding that undesired management activities gets enabled. The rest of the paper is organised as follows. Sections 2 and 3 provide some background and a motivating scenario, respectively. Section 4 illustrates how to test management conformance in multi-component applications. Sections 5 and 6 discuss related work and draw some concluding remarks, respectively.

2 Background: Management Protocols

Topology graphs allow to model multi-component applications [3]. Each node in a topology graph represents an application component, by describing its requirements, the operations to manage it, and the capabilities it features. Arcs then model inter-component dependencies, by associating the requirements of a node to the capabilities of other nodes that are used to satisfy such requirements.

Management protocols [6] describe how the management operations of a node N depend on (i) other operations of the same node N and on (ii) operations of other nodes providing the capabilities that satisfy the requirements of N. The first kind of dependencies is described by a transition relation τ specifying whether an operation o can be executed in a state s, and which state is reached by executing o in s. The second kind of dependencies is described by associating transitions and states with sets of requirements and capabilities. The requirements associated with a transition must be satisfied to allow its execution, while those associated with a state must continue to be satisfied in order for N to continue to work properly. The capabilities associated with a transition or state are those offered by N during such transition or while residing in such state.

Management protocols also specify how N reacts when a fault occurs, i.e. when N is in a state assuming some requirements to be satisfied, but some other node stops satisfying such requirements. This is described by a transition relation φ modelling the fault handling of N by specifying that its state changes from s to s' when some of the requirements it assumes in s stop being satisfied.

Definition 1 (Management protocol). *Let $N = \langle S_N, R_N, C_N, O_N, \mathcal{M}_N \rangle$ be a node, where S_N, R_N, C_N, and O_N are the finite sets of its states, requirements, capabilities, and management operations. $\mathcal{M}_N = \langle \overline{s}_N, \rho_N, \chi_N, \tau_N, \varphi_N \rangle$ is a finite state machine defining the* management protocol *of N, where:*

- $\overline{s}_N \in S_N$ *is the* initial state*,*
- $\rho_N : S_N \rightarrow 2^{R_N}$ *is a function indicating, for each state $s \in S_N$, which conditions on* requirements *must hold,*
- $\chi_N : S_N \rightarrow 2^{C_N}$ *is a function indicating which* capabilities *of N are concretely offered in a state $s \in S_N$,*
- $\tau_N \subseteq S_N \times 2^{R_N} \times 2^{C_N} \times O_N \times S_N$ *is a set of quintuples modelling the* transition relation*, i.e., $\langle s, P, X, o, s' \rangle \in \tau_N$ denotes that in state s, and if condition P holds, o is executable and leads to state s' (by maintaining the capability in X during the transition), and*

– $\varphi_N \subseteq S_N \times S_N$ *is a set of pairs modelling the* fault handling *for a node, i.e.,* $\langle s, s' \rangle \in \varphi_N$ *denotes that the node will change its state from* s *to* s' *if some of the requirements in* $\rho_N(s) - \rho_N(s')$ *stop being satisfied.*

We hereafter assume management protocols to be *complete* (i.e., handling all possible faults in all possible states) and *race-free* (i.e., handling faults so that the simultaneous removal of multiple requirements has the same effect on a node as any sequential removal of the same requirements). Construction rules for ensuring both properties on any management protocol can be found in [6].

3 Motivating Scenario

Consider the (toy) web-based application illustrated in Fig. 1. The application is composed by a gui to which clients connect, and which relies on a backend api to serve them. The api manages application data, by accessing the database where such data is stored. The connections from gui to api and from api to database are represented by arrows connecting a requirement of the source node to a capability of the target node, hence modelling which component is offering the capability used to satisfy a requirement of another component (i.e., the requirement db of api is satisfied by the homonym capability of database, while its capability endp is used to satisfy the homonym requirement of gui). The operations for managing the lifecycle of each component are instead listed next to it.

Suppose that the overall application management has been planned by assuming that the allowed management behaviour for api is that specified by the management protocol of api in Fig. 2(a). The latter indicates that the possible states of api are unavailable (initial), installed, started and failed. Operation transitions allow the component to transit from a state to another by executing the corresponding operation, with configure self-looping on state installed and requiring db to be satisfied for being executable. No requirements are needed to reside in states unavailable and failed, while requirement db is needed to reside in states installed and started (hence requiring database to provide its capability db to satisfy such requirement). If requirement db stops being satisfied while api is installed or started, api gets failed. No requirement is instead needed by api to perform any other transition or reside in any other state. Finally, state started is the only state where api is actually providing its capability endp, hence also being able to satisfy the requirement endp of gui.

Suppose now that we have a candidate implementation of api, provided by a third-party and whose internals are not known. Suppose also that we observed

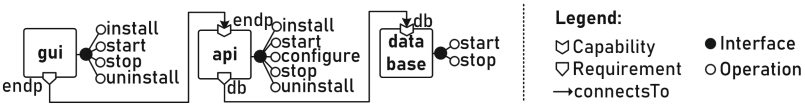

Fig. 1. Example of multi-component application.

that such implementation can be managed according to the management protocol in Fig. 2(b). Can we use the candidate implementation of api in place of its specification? If yes, which guarantees would we get on the overall application management when enacting the replacement? For instance, we may wish to be sure that the overall management behaviour of the application is preserved, so that already developed management plans will continue to work properly.

4 Testing Conformance in Application Management

I/O conformance is usually defined between two IOLTS defining the operational semantics of the formalism under consideration [21]. Thus, before defining management conformance, we first need to introduce an IOLTS semantics for management protocols.

4.1 IOLTS Semantics of Management Protocols

Given a node $N = \langle S_N, R_N, C_N, O_N, \mathcal{M}_N \rangle$, we consider two kinds of *input actions* for each state $s \in S_N$, i.e., *operation-invocation* actions and *requirement-set* actions. An operation-invocation action $o^{\uparrow}$ denotes the input due to the invocation of an operation $o \in O_N$ in a state, while a *requirement-set* action R (with $R \in R_N$) denotes a subset of the requirements of N that are satisfied by capabilities provided by other components in the application.

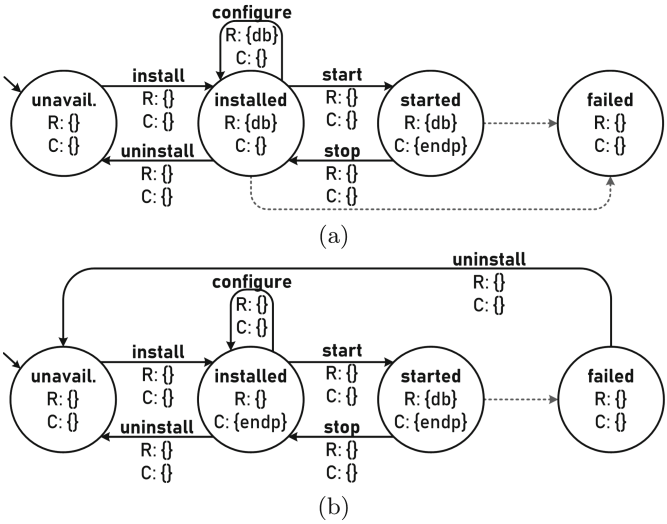

Fig. 2. Examples of management protocols. States are denoted by circles, operation transitions by solid arrows, and fault-handling transitions by dashed arrows. Conditions on requirements and capabilities are specified by sets R and C, respectively.

We instead consider three different kinds of output actions for a node N, for observing outputs possibly occurring after input actions. An *operation termination* action $o^{\downarrow}$ notifies the completion of a previously invoked operation $o \in O_N$. A *capability-set* action allows to denote the set of capabilities that are provided by N to the rest of the application. In addition, a special output symbol $\perp \notin (R_N \cup C_N \cup O_N)$ is used to denote *fault-handling* actions, i.e., to explicitly observe the activation of fault handlers.

Definition 2 (Input/output actions). *Let* $N = \langle S_N, R_N, C_N, O_N, \mathcal{M}_N \rangle$. *The* I/O actions labelling alphabet *is a set* $Act_N = In_N \cup Out_N$ *where*

$$In_N = \{o^{\uparrow} \mid o \in O_N\} \cup 2^{R_N} \quad \textit{and} \quad Out_N = \{o^{\downarrow} \mid o \in O_N\} \cup 2^{C_N} \cup \{\perp\}.$$

We now define the IOLTS semantics of the management protocol $\mathcal{M}_N = \langle \overline{s}_N, \rho_N, \chi_N, \tau_N, \varphi_N \rangle$ of a node N. The configurations X_N of the IOLTS denoting the semantics of $\mathcal{M}_N$ are given by the set of states S_N of N, to which we add a set of fresh configurations denoting the execution of operation and fault-handling transitions, i.e., $X_N = (S_N \cup \tau_N) \cup \varphi_N$. The initial configuration of the IOLTS corresponds to the initial state of $\mathcal{M}_N$, i.e., $\overline{s}_N$.

The transition relation over the configurations of the IOLTS semantics of $\mathcal{M}_N$ are instead obtained as follows.

- For each state $s \in S_N$, two self-looping IOLTS transitions on s indicate the sets of assumed requirements and capabilities provided by N in s.
- For each management protocol transition $t = \langle s, R, C, o, s' \rangle \in \tau_N$, four IOLTS transitions are added. An input transition corresponding to the invocation of o outgoes from s and targets t, while an output transition notifying the completion of o outgoes from t and targets s'. Two transitions self-looping on t instead indicate the sets R and C of requirements and capabilities associated with t, i.e., the input requirement-set and output capability-set.
- For each fault handler $f = \langle s, s' \rangle \in \varphi_N$, and for each subset of requirements assumed in s and handled by $\langle s, s' \rangle$, two IOLTS transitions are added. An input transition labelled with the set R of remaining requirements (i.e., the requirements that were assumed in s and that continue to be satisfied by the rest of the application) goes from s to f, modelling the reaction of N to the handled fault (i.e., the requirements in $\rho_N(s) - R$). An output transition labelled with $\perp$ instead goes from f to s', allowing to explicitly observe the issuing of a fault handler.

Definition 3 (IOLTS semantics). *Let* $N = \langle S_N, R_N, C_N, O_N, \mathcal{M}_N \rangle$ *be a node, with* $\mathcal{M}_N = \langle \overline{s}_N, \rho_N, \chi_N, \tau_N, \varphi_N \rangle$. *The* IOLTS semantics *of the management protocol* $\mathcal{M}_N$ *of* N *is defined as a triple* $\mathcal{I}_N = \langle \overline{s}_N, X_N, \rightarrow_N \rangle$ *where*

$$X_N = S_N \cup \tau_N \cup \varphi_N \quad \textit{and} \quad \rightarrow_N \subseteq (X_N \times Act_N \times X_N),$$

with $\rightarrow_N$ *being the least relation such that*

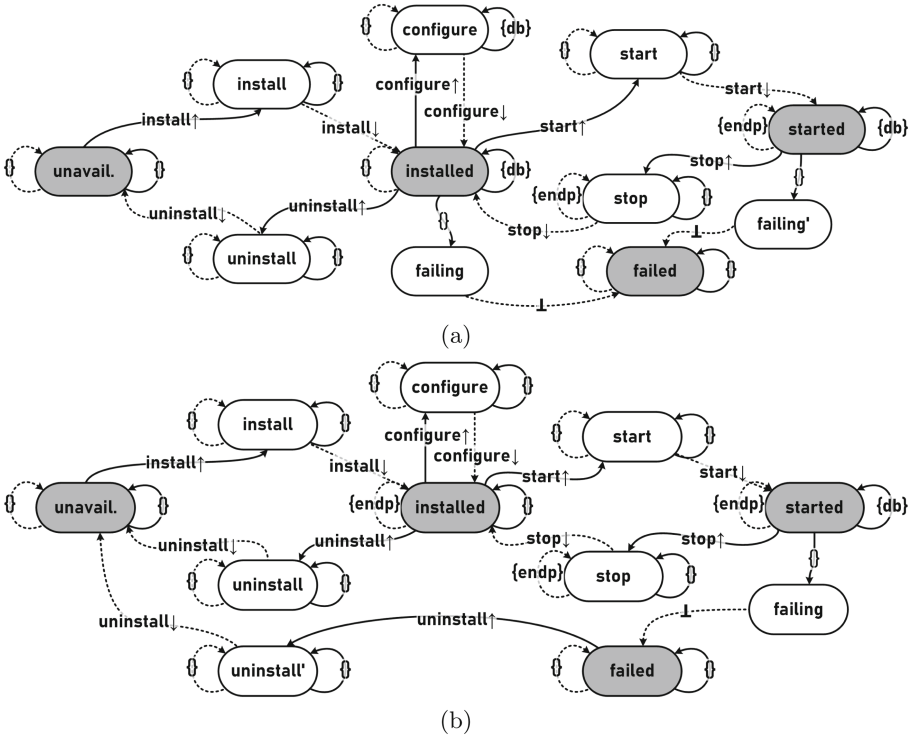

Fig. 3. IOLTS semantics of the management protocols in Fig. 2. Transitions labelled with input actions are solid, while those labelled with output actions are dashed. Configurations corresponding to states of management protocols are highlighted in grey.

- $\forall s \in S_N \,.\, \{\langle s, \rho_N(s), s\rangle, \langle s, \chi_N(s), s\rangle\} \subseteq \rightarrow_N$,
- $\forall t = \langle s, R, C, o, s'\rangle \in \tau_N \,.\, \{\langle s, o^{\uparrow}, t\rangle, \langle t, R, t\rangle, \langle t, C, t\rangle, \langle t, o^{\downarrow}, s'\rangle\} \subseteq \rightarrow_N$,
- $\forall f = \langle s, s'\rangle \in \varphi_N \;\; .$
  $\forall R \subset \rho_N(s) : (\rho_N(s') \subseteq R \wedge (\nexists \langle s, s''\rangle \in \varphi_N \,.\, \rho_N(s') \subset \rho_N(s'') \subseteq R)) \;\; .$
  $\{\langle s, R, f\rangle, \langle f, \bot, s'\rangle\} \subseteq \rightarrow_N .$

Example. Figure 3 illustrates the IOLTS semantics of the management protocols for api in our motivating scenario (Fig. 2). In both IOLTS, configurations are given by the union of the sets of states and transitions of the original management protocol. One can readily observe how intermediate configurations allow to split operation transitions into operation-invocation and operation completion.

Self-loops then model the conditions on requirements and capabilities associated with states and transitions. For instance, the configuration (corresponding to state) started has two self-loops. The dashed self-loop models the fact that the node is offering the capability endp while residing in state started, since it can produce {endp} as output. The solid self-loop instead indicates that the node keeps residing in state started if the requirement db continues to be satisfied, since the configuration does not change when {db} is given as input.

The figure also shows how fault-handling transitions are split into two transitions. Consider again started, whose corresponding state assumes db to be satisfied. If no requirement is given as input, this means that db stops being satisfied and the configuration of the IOLTS changes from started to failing, from which (the configuration corresponding to) state failed can be reached by producing the output $\perp$. The two transitions in the IOLTS model the corresponding fault-handling transition in the original management protocol.

4.2 Input-Enabledness

A crucial assumption in I/O conformance testing is *input-enabledness* of implementations under test, i.e., a candidate implementation under test will never block any input action [21]. During our case, this means that a management protocol is input enabled if its IOLTS semantics accepts any possible input in any configuration corresponding to a state in the original management protocol.

Notation. *Given the IOLTS semantics* $\mathcal{I}_N = \langle \overline{s}_N, X_N, \rightarrow_N \rangle$ *of a management protocol* $\mathcal{M}_N$ *and a configuration* $x \in X_N$, $x \xrightarrow{\sigma}_N x'$ *and* $x \xrightarrow{\sigma}_N$ *(with* $\sigma \in Act_N^*$*) denote traces* σ *corresponding to valid paths in* $\mathcal{I}_N$.

Definition 4 (Input-enabledness). *Let* $\mathcal{I}_N = \langle \overline{s}_N, X_N, \rightarrow_N \rangle$ *be the IOLTS semantics of the management protocol of a node* N. $\mathcal{I}_N$ *is* input-enabled *iff*

$$\forall x \in S_N \,.\, \forall i \in In_N : x \xrightarrow{i} .$$

The input-enabledness of a given management protocol can be ensured automatically. Intuitively, its IOLTS semantics can be automatically completed by adding an input transition targeting a distinct sink configuration $s_\perp$ (with $s_\perp \notin X_N$) for each unspecified input of each state $s \in S_N$. Namely, an input transition labelled with the set R of requirements is added if there is no input transition outgoing from s and labelled with R. An input transition labelled with the invocation of operation o is instead added if o cannot be invoked in s.

The sink state $s_\perp$ is also made input-enabled, by adding a self-looping input transition for each possible input. Furthermore, a self-looping output transition on $s_\perp$ is added, which is labelled with the special symbol $\perp$. This allows to explicitly observe that an unspecified input has been provided to the IOLTS, as whenever this happens the IOLTS can provide $\perp$ as output. Any unspecified input action hence results in an (implicit) fault handling under input completion.
Example (cont.). Consider again the (a) management protocol specification and (b) candidate implementation in our motivating example (Fig. 2). By looking at their corresponding IOLTS semantics, shown in Fig. 3, one can readily observe that both management protocols are not input-enabled, as each configuration corresponding to a state of the protocol lacks some outgoing input transitions. More precisely, there are some operation-invocation actions that are available in each of such configurations, e.g., in the IOLTS semantics of both protocols there is only one out of five operation-invocation actions defined for started.

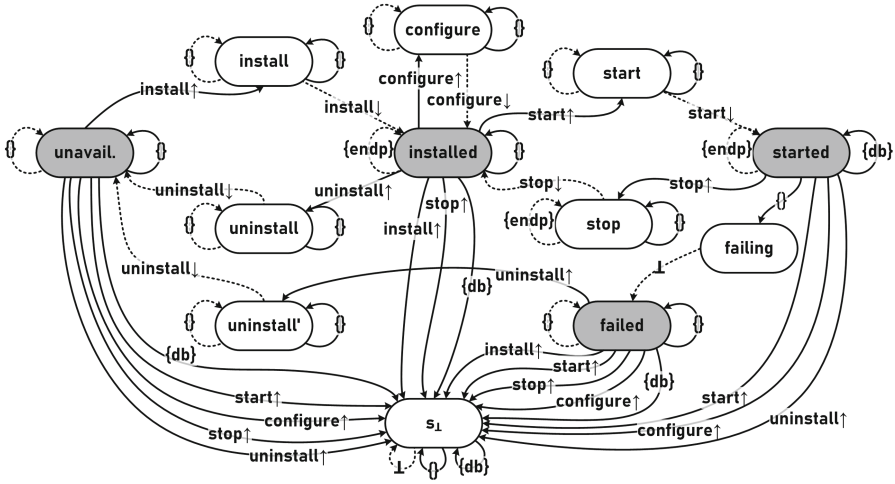

Fig. 4. Input-enabled version of the IOLTS in Fig. 3(b), obtained by input completion.

I/O conformance testing requires implementations to be input-enabled [21]. To be able to test whether the candidate implementation of api conforms to its specification, we hence need to make the IOLTS in Fig. 3(b) input-enabled. The latter can be obtained by applying it to the above listed construction rules, which results in the IOLTS shown in Fig. 4.

4.3 Conformance Testing Based on Management Protocols

We now introduce a formal framework for testing management conformance between application components. Suppose N to constitute the (fully known) *specification* of the intended management behaviour of an application component and N' to constitute a candidate (black-box) *implementation* for N, both defined over the same sets of requirements, capabilities and operations. Intuitively, we wish to formally define an I/O conformance relation on management protocols (mpioco), such that N' mpioco N denotes that the management behaviour implemented by N' *conforms* to that specified by N, i.e., given the same inputs, N' can produce the outputs specified by N. Given how we relate input and outputs to requirements, operations and capabilities (Definition 2), this intuitively means that given the same requirements and operations, N' can offer the capabilities expected in N, hence meaning that N' can be used to replace N in a multi-component application.

For defining mpioco, we first need to introduce the notions of *quiescence* and *suspension traces*. We introduce a special output symbol $\delta \notin (R_N \cup C_N \cup O_N)$ denoting *quiescence*, i.e., the observable absence of any output. In the IOLTS semantics of a management protocol, *quiescence* occurs whenever a configuration corresponds to a state not providing any capability. Suspension traces are then defined by extending existing traces and allowing to explicitly observe quiescence.

Definition 5 (Quiescence and suspension traces). *Let $N = \langle S_N, R_N, C_N, O_N, \mathcal{M}_N \rangle$ be a node and let $\mathcal{I}_N = \langle \overline{s}_N, X_N, \rightarrow_N \rangle$ be the IOLTS semantics of $\mathcal{M}_N$. Let also $x \in X_N$ be a configuration in $\mathcal{I}_N$.*
– x is quiescent*, denoted by $\delta(x)$ iff $\forall C \subseteq C_N : C \neq \emptyset \,.\, x \not\xrightarrow{C}_N$, and*
– $\mathsf{straces}(x) = \{\sigma \mid x \xrightarrow{\sigma}_N\}$, where $\forall x' \in X_N \,.\, x' \xrightarrow{\delta}_N x'$ if $\delta(x')$.

We also need to introduce the notions of *enabled outputs* and *reachability*, to identify the set of output symbols enabled by a set of configurations, and the configurations that can be reached by performing a trace σ in a configuration x.

Definition 6 (Enabled outputs). *Let $N = \langle S_N, R_N, C_N, O_N, \mathcal{M}_N \rangle$ be a node and let $\mathcal{I}_N = \langle \overline{s}_N, X_N, \rightarrow_N \rangle$ be the IOLTS semantics of $\mathcal{M}_N$. The set of* outputs *enabled in a configuration $x \in X_N$ is the least set $\mathsf{out}(x)$ such that*

$$C \subseteq \mathsf{out}(x) \text{ if } x \xrightarrow{C} \quad \text{and} \quad \{\bot\} \subseteq \mathsf{out}(x) \text{ if } x \xrightarrow{\bot} \quad \text{and} \quad \{\delta\} \subseteq \mathsf{out}(x) \text{ if } \delta(s).$$

We also write $\mathsf{out}(X)$ to denote the outputs enabled in at least one of the configurations in the set $X \subseteq X_N$, i.e., $\mathsf{out}(X) = \bigcup_{x \in X} \mathsf{out}(x)$.

We define two different versions of reachability, distinguished by parameter γ. If γ is "=", transitions involving a set of requirements or capabilities are considered only if the trace is delivering exactly that set of requirements or capabilities. In the relaxed version with γ set to "$\geq$", a transition labelled with a set R' of requirements is considered if the trace is delivering at least the requirements in R', while one labelled with a set C' of capabilities is considered if the trace is delivering at most the capabilities in C'. Operation-invocation and operation-completion transitions are instead always considered, independently of γ.

Definition 7 (γ-reachability). *Let $N = \langle S_N, R_N, C_N, O_N, \mathcal{M}_N \rangle$ be a node and let $\mathcal{I}_N = \langle \overline{s}_N, X_N, \rightarrow_N \rangle$ be the IOLTS semantics of $\mathcal{M}_N$. Let also $x \in X_N$ be a configuration in $\mathcal{I}_N$ and $\sigma \in \mathsf{straces}(x)$ be a suspension trace for x. The set of* γ-reachable *configurations from x with σ is $\mathsf{reach}_\gamma(x, \sigma) \subseteq X_N$, i.e., the least set satisfying the following recursive rules:*

$$-x \in \mathsf{reach}_\gamma(x, \epsilon),$$
$$-x \in \mathsf{reach}_=(x', \sigma) \text{ if } x' \xrightarrow{R'}_N x'' \wedge R' \subseteq R_N \wedge \sigma = R' \cdot \sigma'' \wedge x \in \mathsf{reach}_=(x'', \sigma''),$$
$$-x \in \mathsf{reach}_\geq(x', \sigma) \text{ if } x' \xrightarrow{R'}_N x'' \wedge R' \subseteq R \subseteq R_N \wedge \sigma = R \cdot \sigma'' \wedge x \in \mathsf{reach}_\geq(x'', \sigma''),$$
$$-x \in \mathsf{reach}_=(x', \sigma) \text{ if } x' \xrightarrow{C'}_N x'' \wedge C' \subseteq C_N \wedge \sigma = C' \cdot \sigma'' \wedge x \in \mathsf{reach}_=(x'', \sigma''),$$
$$-x \in \mathsf{reach}_\geq(x', \sigma) \text{ if } x' \xrightarrow{C'}_N x'' \wedge C \subseteq C' \subseteq C_N \wedge \sigma = C \cdot \sigma'' \wedge x \in \mathsf{reach}_\geq(x'', \sigma''),$$
$$-x \in \mathsf{reach}_\gamma(x', \sigma) \text{ if } x' \xrightarrow{o^\uparrow} x'' \wedge o \in O_N \wedge \sigma = o^\uparrow \cdot \sigma'' \wedge x \in \mathsf{reach}_\gamma(x'', \sigma''), \text{ and}$$
$$-x \in \mathsf{reach}_\gamma(x', \sigma) \text{ if } x' \xrightarrow{o^\downarrow} x'' \wedge o \in O_N \wedge \sigma = o^\downarrow \cdot \sigma'' \wedge x \in \mathsf{reach}_\gamma(x'', \sigma''),$$

where $\cdot$ denotes concatenation (i.e., $\alpha \cdot \omega$ denotes an I/O action α followed by a sequence ω of I/O actions).

We now formally define how to test the management conformance between application components, by means of a parametric relation for testing management conformance, which allows to obtain four different testing operators. The latter are distinguished based on (a) the employed notion of γ-reachability and on (b) the way non-deterministic output-behaviour is handled. Concerning (b), we introduce some shorthand notations for comparing sets of sets, i.e., we write $Z' \sqsupset Z''$ to indicate that Z'' contains all sets in Z', and $Z' \sqsupseteq Z''$ to indicate that Z'' contains a superset of each set in Z'.

Notation. *Let Z' and Z'' be two sets. We write $Z' \sqsupset Z''$ iff $\forall z' \in Z' : (\exists z'' \in Z'' : z'' = z')$, and $Z' \sqsupseteq Z''$ iff $\forall z' \in Z' : (\exists z'' \in Z'' : z'' \subseteq z')$.*

Intuitively, the implemented management of a node conforms to its specification if, given a set of inputs, it can produce the expected outputs.

Definition 8 (mpioco). *Let $N = \langle S_N, R_N, C_N, O_N, \mathcal{M}_N \rangle$ be a node, with $\mathcal{M}_N = \langle \overline{s}_N, \rho_N, \chi_N, \tau_N, \varphi_N \rangle$. Let $N' = \langle S'_N, R_N, C_N, O_N, \mathcal{M}'_N \rangle$ be another node, with $\mathcal{M}_N = \langle \overline{s}'_N, \rho'_N, \chi'_N, \tau'_N, \varphi'_N \rangle$ being input-enabled. Let also $\beta \in \{\sqsupset, \sqsupseteq\}$ and $\gamma \in \{=, \geq\}$.*

$$N' \,\mathsf{mpioco}_{\beta,\gamma}\, N \Leftrightarrow \forall \sigma \in \mathsf{straces}(\overline{s}_N)\,.\,\mathsf{out}(\mathsf{reach}_\gamma(\overline{s}'_N, \sigma))\,\beta\,\mathsf{out}(\mathsf{reach}_\gamma(\overline{s}_N, \sigma)).$$

Note that if γ is $=$ and β is $\sqsupset$, an implementation conforms to a specification only if it produces the set of desired capabilities given exactly the same sets of requirements. Setting γ to $\geq$ or β to $\sqsupset$ results in more flexibile relations of management conformance. With γ set to $\geq$, conformance occurs also if the implementation needs less requirements and provides more capabilities. With β set to $\sqsupseteq$, conformance occurs also if the implementation can produce at least one of the expected sets of capabilities, hence allowing specifications to exhibit non-deterministic output behaviour.

Example (cont.). Consider again our motivating scenario, where we have a specification of api and a possible implementation for such component, which we hereafter denote with api_S and api_I, for simplicity. The IOLTS semantics of the management protocol of api_S is in Fig. 3(a), while the input-complete IOLTS semantics of the management protocol of api_I is in Fig. 4.

By applying the different conformance testing relations to both IOLTS, one can check that $\mathsf{api}_I \,\mathsf{mpioco}_{\sqsupset,\geq}\, \mathsf{api}_S$ and $\mathsf{api}_I \,\mathsf{mpioco}_{\sqsupseteq,\geq}\, \mathsf{api}_S$ (while the same does not hold for the relations with γ set to $=$). This means that the candidate implementation api_I can be used to replace the desired specification api_S. However, which guarantees on the overall application management are given when enacting the replacement? Is every possible trace of management preserved? Is there any (potentially undesired) additional trace that gets enabled?

4.4 Which Conformance Tests to Run?

The different notions of mpioco not only allow to check management conformance, but they can also ensure different properties while replacing a component

Table 1. Additional guarantees on overall application management, after replacing a specification with a conforming implementation, with mpioco varying on β and γ.

	γ **set to** $=$	γ **set to** $\geq$
β **set to** $\sqsupset$	*existing traces preserved* *no additional traces*	*existing traces preserved*
β **set to** $\sqsupseteq$	*no additional traces*	–

specification with a conforming implementation. More precisely, mpioco relations vary based on the parameters γ and β, and stricter mpioco relations are obtained when employing stricter constraints on γ and β, i.e., setting γ to $=$ and β to $\sqsupset$. Both restrictions induce additional guarantees on the overall management behaviour of a multi-component application (Table 1).

Whenever the implementation of a node conforms to a specification with γ set to $=$, this means that the implementation needs the same sets of requirements in states and transitions, and that it provides the same sets of capabilities. As a result, after replacing the specification with the conforming implementation in an applicaton, *no additional trace* in the overall management behaviour is introduced. This intuitively holds since the execution of operations and fault handlers is constrained by conditions on requirements and capabilities, which do not change after enacting the replacement.

On the other hand, the implementation of a node can conform to a specification with β set to $\sqsupset$ only if such specification is deterministic in its output behaviour. The latter happens only if the specification does not contain non-deterministic branches, and (given Definition 8) it can be proved that this means that any conforming implementation implements all its viable paths. This in turn means that setting β set to $\sqsupset$ results in *preserving all possible traces* in the overall management behaviour of an application.

Which of the restrictions to employ strictly depends on the guarantees that an application administrator wishes to have. If an application administrator wishes to replace the specification of a component by preserving the overall management behaviour of the application it appears in, she must test the management conformance of a candidate implementation with β set to $\sqsupset$. The latter was precisely the case in our motivating scenario, and since the candidate implementation of api shown to be conforming to the desired specification with β and γ set to $\sqsupset$ and $\geq$, we can use such implementation to replace the given specification.

Alternatively, if an application administrator wishes to replace the specification of a component by ensuring that no additional management trace is introduced, she has to test for management conformance with γ set to $=$. This is not to be underestimated, as enabling additional management activities while considering interdepedent components may result in some undesired situation. For instance, suppose that a component specification requires a VPN in some state to encrypt its communications. By employing the relaxed version of γ, an implementation not requiring any VPN would conform the specification, even

if it this would mean that after enacting the replacement the component would not be exploiting any VPN to encrypt its communications.

5 Related Work

Various approaches allow to check whether an existing implementation can be used to replace the specification of a desired application component, e.g., [4], [5], [9], [11], and [12] just to mention some. Such approaches typically consider an implementation as suitable to replace a specification if the implementation can provide (at least) the desired outputs if provided with (at most) the same inputs, by also providing techniques for adapting matching implementations to exhibit the specified I/O behaviour. Their goal is indeed to enact the replacement of a component specification with a suitable implementation, by ensuring that the overall application behaviour is preserved, which in our case can be obtained by exploiting mpioco with β restricted to $\sqsupset$. Our approach is instead intended to support application administrators in a wider set of scenarios, varying on the guarantees she wishes to get on the overall management of an application.

Similar arguments apply to the approach proposed in [17]. The latter propose an approach for checking that the interactions with a service in a multi-service application (including the handling of potential exceptions) conforms the behaviour specified by the service itself, hence focusing on preserving the overall application behaviour. Our approach applies to a wider set of scenarios, depending on desired guarantees on the overall application management behaviour.

To offer such a wider support, we exploit the potentials of Tretmans' I/O conformance testing theory [21]. There exists various heterogeneous extensions and variations of the I/O conformance testing theory, and the closest to ours are those dealing with (i) the implementation freedom given by specifications with non-deterministic output behaviour, with (ii) fault handling, and with (iii) guarantees on the overall application behaviour after replacing a component specification with a conforming implementation.

Approaches worth mentioning for what concerns implementation freedom are [2,10,18]. [2] extends I/O conformance testing for dealing with software product lines, by allowing them to exhibit a fine-grained behavioral variability controlled by feature selection. [10,18] give implementation freedom by introducing modality in I/O conformance testing, i.e., allowing to distinguish between mandatory and optional output behaviour. Various other existing approaches define modal conformance as alternating simulation relations [1], where conformance is lifted from simple trace inclusion to an alternating simulation preorder [16,22]. However, none of the above approaches allows to capture the implementation freedom characterising conformance testing on management protocols, e.g., allowing an implementation to conform to a given specification even if the former needs less requirements or provides more capabilities.

To the best of our knowledge, ours is also the first approach for testing conformance for software systems with explicit fault-handling. Only [19,20] consider explicit failure states, but for different purposes. They indeed consider failure

states as forbidden states, to suspend test runs in case of forbidden inputs. However, [19,20], as well as no other approach for conformance testing, currently support the explicit specification and testing of fault-handling mechanisms such as those provided by management protocols.

In summary, to the best of our knowledge, ours is the first approach for testing conformance of the *management* allowed by the implementation of a component with respect to that of its specification. Our approach distinguishes from existing solutions for checking behaviour-aware replaceability in terms of supported scenarios, enabled by the proposed relation of conformance testing. The latter is itself the first relation providing the freedom to implement a specification by requiring less and offering more, as well as dealing with explicit fault-handling and with different operators for combining the behaviour of application components.

6 Conclusions

We have presented an approach for testing management conformance in multi-component applications. More precisely, we proposed a parametric relation for testing whether the management allowed by an existing component conforms to a desired specification, modelled with management protocols [6]. Our parametric relation can be instantiated into four different conformance testing relations, spanning from that giving higher implementation freedom, to more restricting relations ensuring that replacing a specification with a conforming implementation continues to enable all already allowed management activities, and/or that no additional (potentially undesired) management activity gets enabled.

We also discussed how the different conformance testing relations can be used to check the replaceability of the specification of a component with a conforming implementation, and how the choice of which relation to exploit strictly depends on the desiderata of an application administrator. She may decide to reduce implementation freedom (hence restricting the set of implementations conforming to a given specification), if she wishes to ensure that the overall application management is fully preserved after replacing a specification with a conforming implementation, or that no undesired management activity gets enabled.

We now plan to provide a first prototype for testing management conformance in multi-component applications and to use such a prototype to validate our approach in practice. We also plan to extend the supported conformance tests, by relying on more expressive versions of management protocols (e.g., truly concurrent management protocols [7]), and by extending the degree of implementation freedom (e.g., by introducing modality, as [10,18] do for different purposes). We also plan to investigate whether and how to adapt our conformance testing approach to other approaches for modelling the management of multi-component applications, e.g., the Aeolus component model [13].

Acknowledgments. This work is partly funded by the projects *AMaCA* (POR-FSE, Regione Toscana) and *DECLware* (PRA_2018_66, University of Pisa). This work was funded by the Hessian LOEWE initiative within the Software-Factory 4.0 project.

References

1. Alur, R., Henzinger, T.A., Kupferman, O., Vardi, M.Y.: Alternating refinement relations. In: Sangiorgi, D., de Simone, R. (eds.) CONCUR 1998. LNCS, vol. 1466, pp. 163–178. Springer, Heidelberg (1998). https://doi.org/10.1007/BFb0055622
2. Beohar, H., Mousavi, M.R.: Input-output conformance testing for software product lines. J. Log. Algebr. Methods Program. **85**(6), 1131–1153 (2016)
3. Binz, T., Fehling, C., Leymann, F., Nowak, A., Schumm, D.: Formalizing the cloud through enterprise topology graphs. In: 2012 IEEE Fifth International Conference on Cloud Computing, pp. 742–749. IEEE (2012)
4. Bonchi, F., Brogi, A., Canciani, A., Soldani, J.: Simulation-based matching of cloud applications. Sci. Comput. Program. **162**, 110–131 (2018)
5. Bonchi, F., Brogi, A., Corfini, S., Gadducci, F.: A net-based approach to web services publication and replaceability. Fundam. Inform. **94**(3–4), 305–330 (2009)
6. Brogi, A., Canciani, A., Soldani, J.: Fault-aware management protocols for multi-component applications. J. Syst. Softw. **139**, 189–210 (2018)
7. Brogi, A., Canciani, A., Soldani, J.: True concurrent management of multi-component applications. In: Kritikos, K., Plebani, P., de Paoli, F. (eds.) ESOCC 2018. LNCS, vol. 11116, pp. 17–32. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-99819-0_2
8. Brogi, A., Rinaldi, L., Soldani, J.: TosKer: a synergy between TOSCA and Docker for orchestrating multicomponent applications. Soft. Pract. Exp. **48**(11), 2061–2079. https://doi.org/10.1002/spe.2625
9. Brogi, A., Soldani, J.: Finding available services in TOSCA-compliant clouds. Sci. Comput. Program. **115–116**, 177–198 (2016)
10. Bujtor, F., Sorokin, L., Vogler, W.: Testing preorders for dMTS: deadlock-and the new deadlock-/divergencetesting. ACM Trans. Embed. Comput. Syst. **16**(2), 41:1–41:28 (2016)
11. Castagna, G., Gesbert, N., Padovani, L.: A theory of contracts for web services. ACM Trans. Program. Lang. Syst. **31**(5), 19:1–19:61 (2009)
12. Cavallaro, L., Di Nitto, E., Pradella, M.: An automatic approach to enable replacement of conversational services. In: Baresi, L., Chi, C.-H., Suzuki, J. (eds.) ICSOC/ServiceWave -2009. LNCS, vol. 5900, pp. 159–174. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-10383-4_11
13. Di Cosmo, R., Mauro, J., Zacchiroli, S., Zavattaro, G.: Aeolus: a component model for the cloud. Inf. Comput. **239**, 100–121 (2014)
14. Dragoni, N., et al.: Microservices: yesterday, today, and tomorrow. Present and Ulterior Software Engineering, pp. 195–216. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-67425-4_12
15. Durán, F., Salaün, G.: Robust and reliable reconfiguration of cloud applications. J. Syst. Softw. **122**, 524–537 (2016)
16. Gregorio-Rodríguez, C., Llana, L., Martínez-Torres, R.: Input-output conformance simulation (iocos) for model based testing. In: Beyer, D., Boreale, M. (eds.) FMOODS/FORTE -2013. LNCS, vol. 7892, pp. 114–129. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-38592-6_9
17. Heike, C., Zimmermann, W., Both, A.: On expanding protocol conformance checking to exception handling. Serv. Oriented Comput. Appl. **8**(4), 299–322 (2013). https://doi.org/10.1007/s11761-013-0146-2
18. Luthmann, L., Mennicke, S., Lochau, M.: Towards an I/O conformance testing theory for software product lines based on modal interface automata. In: Formal Methods and Analysis in SPL Engineering. EPTCS, vol. 182, pp. 1–13 (2015)

19. Luthmann, L., Mennicke, S., Lochau, M.: Compositionality, decompositionality and refinement in input/output conformance testing. In: Kouchnarenko, O., Khosravi, R. (eds.) FACS 2016. LNCS, vol. 10231, pp. 54–72. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-57666-4_5
20. Luthmann, L., Mennicke, S., Lochau, M.: Unifying modal interface theories and compositional input/output conformance testing. Sci. Comput. Program. **172**, 27–47 (2019)
21. Tretmans, J.: Test generation with inputs, outputs and repetitive quiescence. Soft. Concepts Tools **17**(3), 103–120 (1996)
22. Veanes, M., Bjørner, N.: Input-output model programs. In: Leucker, M., Morgan, C. (eds.) ICTAC 2009. LNCS, vol. 5684, pp. 322–335. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-03466-4_21