On the Evolution of Reactive Components – A Process-Algebraic Approach –

Markus Müller-Olm[1], Bernhard Steffen[1], and Rance Cleaveland[2]

[1] Dept. of Comp. Sci., University of Dortmund, 44221 Dortmund, Germany
{mmo,steffen}@cs.uni-dortmund.de
[2] Dept. of Comp. Sci., SUNY at Stony Brook, Stony Brook, NY 11794-4400, USA
rance@cs.sunysb.edu

Abstract. A common problem in library-based programming is the *downward compatibility* problem: will a program using an existing version of a library continue to function correctly with an upgraded version? As a step toward addressing this problem for libraries of *reactive* components we develop a theory that equips components with *interface languages* characterizing the interaction patterns user applications may engage in with the component. We then show how these languages may be used to build *upgrade specifications* from components and their interface languages. Intuitively, upgrade specifications explicitly describe requirements an (improved) implementation of a component must satisfy and are intended for use by library developers. Under certain reasonable assumptions about the contexts components are to be used in we show that our upgrade specifications are complete in the sense that every correct upgrade of a component is related in a precise manner to its upgrade specification. In particular, these results hold if the language being used to develop contexts is CSP or CCS.

Keywords: action transducer, bisimulation, context, interface language, downward compatibility, process-algebra, refinement.

1 Introduction

Practical software development relies heavily on the use of *libraries* of previously implemented components. By allowing the cost of module development to be amortized over the number of systems that use the modules, libraries contribute to substantially cheaper software. As library components are also subjected to more rigorous validation by virtue of their inclusion in different systems, using them judiciously can also improve the reliability of systems. These obvious benefits have led to a profusion of libraries in a variety of different application areas in programming.

A common problem in library-based programming is the *upgrade* (or *downward compatibility*) problem: given an "improved" version of an existing library, will applications using the existing version of the library continue to function correctly without modification? Users and implementors would clearly wish this

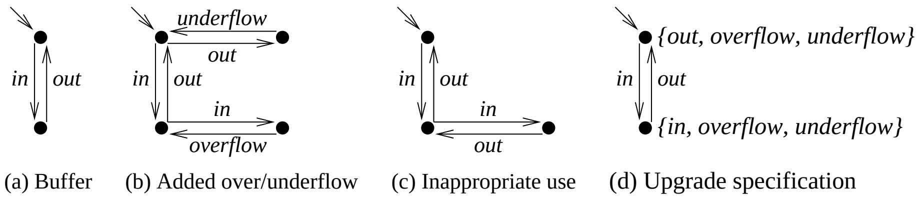

Fig. 1. A one-place buffer and its "upgrade".

to be the case, and yet anyone who has maintained a program knows that this rarely holds.

A partial solution in widespread use in the realm of sequential programming relies on the use of component *types* as "interface specifications" (one need only consider the myriad of `.h` files in existence to see the prevalence of such specifications). Clearly, preservation of types in an upgrade is far too weak to guarantee full downward compatibility, as two procedures may behave quite differently even though they share the same type. If, however, we focus solely on the interaction pattern between components and applications, types indeed provide "full" information: a user need only know the types of parameters and results to interact correctly with a procedure, as a procedure's "interaction pattern" consists of strictly alternating sequences of calls and returns.

The situation for *reactive* components is different. Such components are intended to maintain an ongoing interaction with the system in which they are used, as in control software (e.g. for avionics systems) or user interfaces (e.g. window managers). In this case traditional type information is clearly inadequate, as a component can input and output several times during its execution. Even a component's alphabet of input and output actions, which may be viewed as analogous to type information, says little about allowed interaction patterns, as the next paragraph shows.

As a small but motivating example of the subtle problems arising in upgrades of reactive libraries, consider a library containing an implementation of a one-place buffer as pictured in Fig. 1(a). The programmer's system accesses this buffer via the actions *in* (for input) and *out* (for output), with the buffer initially being empty.

Suppose users complain that a one-place buffer reporting under- and overflows would be far more convenient. In order to address these complaints the maintainer of the library might decide to enhance its functionality by providing the innocuously looking upgrade shown in Fig. 1(b). The question now confronting the programmer and the maintainer is this: can systems constructed using the old buffer component safely use the new implementation of the buffer? As the alphabet of the new component includes that of the old, one would be tempted to say "yes"; however, note that the system in Fig. 1(c) can lead to deadlock when connected with the new component but not with the old. Traditional results of process algebra are also of little use in answering this question, since no reasonable notion of semantic equivalence or refinement would relate the new buffer to the old one. Indeed, in the absence of any information about the context in which the buffer is being used, the only safe answer to this question would be "no".

On the other hand, suppose that the developer of the original buffer module equipped it with the regular *interface language* $(in.out)^*.(in + \varepsilon)$ expressing the (otherwise implicit) assumption that a user could assume a one-place buffer behavior *only if* his system neither tries to store more than one element nor to extract an element from an empty buffer. The system in Fig. 1(c) does not obey this interface language and hence one would not expect to be able to use the buffer upgrade with this system. An upgrade of a module providing additional functionality could have an enlarged interface language in order to allow access to the new capabilities in future systems. The new buffer, e.g., could be equipped with the extended interface language $(in + out + underflow + overflow)*$. A smaller language might be chosen in order to preserve more potential for future upgrades.

In this paper we study the utility of furnishing reactive components with interface information in the form of such (prefix-closed) interface languages containing the sequences of input and output actions that applications are permitted to exchange with the component. Given such interface information, we define what it means for an upgrade to be correct, and we show that from an interface and a component, one can characterize the correct upgrades of a component via a single labeled-transition-system-like specification. These specifications, which we term *upgrade specifications*, are of particular use to library developers, as they clearly indicate what information in a component must be preserved in order for a component to be downward compatible.

The remainder of the paper develops along the following lines. The next section presents the operational models of components and contexts that we use in our technical development. Section 3 defines the space of upgrade specifications and the next section establishes the main result of this paper, namely, that the correct replacements of a component given a certain interface language can be characterized by a single upgrade specification. The section following then shows that our results hold even for restricted classes of contexts such as the parallel contexts of CCS [9] or CSP [4], and the final section contains our conclusions and directions for future research and discusses related work.

2 Processes and Contexts

Processes. We model processes (components) by labeled transition systems with a designated start state. Formally, a process is a quadruple $P = (S, A, \rightarrow, p_0)$, where S is a set of *states*, A is an *alphabet of actions* that the process might exchange with its environment in a single computation step, $\rightarrow \subseteq S \times A \times S$ is a *transition relation*, and $p_0 \in S$ is the *start state*. We denote the set of processes with action alphabet A by Proc_A and the letters P and Q range over processes.

Given a process P, we write $S_P, A_P, \rightarrow_P$ and p_0 to refer to its state set, alphabet of actions, transition relation, and start state, respectively, and use (possibly decorated versions of) the corresponding lower case letter – p in this case – to range over S_P. P is called *finite-state* if both S_P and A_P are finite. The relationship $(p, a, p') \in \rightarrow$, for which we write $p \xrightarrow{a} p'$ in the following, indicates that in state p, P can evolve to state p' under the observation of a. We generalize the transition relation $\rightarrow$ to words $w \in A^*$ by the usual inductive definition:

$$p \xrightarrow{\varepsilon} p' \text{ iff } p = p' \qquad \text{and} \qquad p \xrightarrow{w \cdot a} p' \text{ iff } \exists p'' : p \xrightarrow{w} p'' \wedge p'' \xrightarrow{a} p' \ ,$$

where ε is the empty word, $w \in A^*$, and $a \in A$. The language of a state $p \in S$ is the set $L(p) = \{w \mid \exists p' : p \xrightarrow{w} p'\}$. The language $L(P)$ of process P is the language $L(p_0)$ of its initial state.

A state $p \in S$ is called *reachable* if there is a word $w \in A^*$ such that $p_0 \xrightarrow{w} p$ and a process is called *deterministic* if for any reachable $p \in S$ and $a \in A$ at most one $p' \in S$ exists such that $p \xrightarrow{a} p'$.

Contexts. We adopt the framework of Larsen and Xinxin [7, 8] and use *action transducers* as basic operational model of contexts. The idea is to interpret contexts as special transition systems that *consume* actions from the inner *parameter processes* and produce actions for the environment. For the purpose of this paper only unary contexts are needed, i.e. contexts having just one parameter process. Formally, a *context* is a structure $C = (S, A, B, \rightarrow, c_0)$, where S is a set of *context states*, A is the action alphabet of the parameter process, B is the action alphabet of the resulting process, $\rightarrow \subseteq S \times (A \cup \{0\}) \times B \times S$ is the *transduction relation*, and c_0 is the *start state* of the context. Here 0 is assumed to be a distinguished non-action symbol; in particular, $0 \notin A$.

The letter C ranges over contexts and we assume, as for processes, that the constituting parts of a context C can be referenced by S_C, A_C etc. Moreover, c and variables derived from c by decoration range over S_C. The set of contexts for processes with alphabet A is $\mathsf{Ctx}_A = \{C \mid C \text{ is a context and } A_C = A\}$. Note that we allow contexts with different inner and outer alphabet and assume a designated start state in contrast to Larsen and Xinxin in [7, 8].

We often use the intuitive notation $c \xrightarrow[a]{b}_C c'$ in favor of $(c, a, b, c') \in \rightarrow_C$ and call this a *transduction*. A transduction where $a \neq 0$ is interpreted as follows: if the context is currently in state c it can "consume" an a-labeled transition from the parameter process and evolve to its state c', producing action b in doing so. A transduction with $a = 0$ represents an autonomous step of the context, i.e. a step without an interaction with the parameter process. Formally this intuition is captured by the following definition.

Definition 1 (Context application). *Suppose that C and P are such that $A_C = A_P$. Then $C(P)$ is the process $(S_C \times S_P, B_C, \rightarrow, c_0(p_0))$, where the transition relation $\rightarrow$ is the smallest relation obeying the following two rules:*

$$\frac{p \xrightarrow{a}_P p' \ , \ c \xrightarrow[a]{b}_C c'}{c(p) \xrightarrow{b} c'(p')} \qquad\qquad \frac{c \xrightarrow[0]{b}_C c'}{c(p) \xrightarrow{b} c'(p)}$$

Here and in the following we use the notation $c(p)$ for pairs $(c, p) \in S_C \times S_P$.

It is well-known that transition systems are a rather fine-grained model of processes. Therefore various equivalences have been studied in the literature that identify processes on the basis of their behavior. In this paper we consider the classic notion of (strong) bisimulation [10, 9].

Definition 2 (Bisimulations). *Suppose P, Q are processes with same action alphabet, i.e. $A_P = A_Q$. A relation $R \subseteq S_P \times S_Q$ is called a* bisimulation *if for all $(p, q) \in R$ the following two conditions hold:*

- $\forall a, p' : p \xrightarrow{a}_P p' \Rightarrow \exists q' : q \xrightarrow{a}_Q q' \wedge (p', q') \in R$, *and*
- $\forall a, q' : q \xrightarrow{a}_Q q' \Rightarrow \exists p' : p \xrightarrow{a}_P p' \wedge (p', q') \in R$.

Define $\sim$ to be the union of all bisimulations R. The processes P and Q are called bisimilar*, $P \sim Q$ for short, if $p_0 \sim q_0$.*

It is well-known that $\sim$ is the largest bisimulation on $S_P \times S_Q$. It is also straightforward to prove that any context preserves bisimilarity, i.e. that $P \sim Q$ implies $C(P) \sim C(Q)$.

Interface Languages. The language consisting of all words in A_C^* that a context C potentially exchanges with parameter processes is called its *interface language*. Its formal definition is based on the straightforward inductive extension of the transduction relation $\rightarrow_C$ to words:

$$c \xrightarrow[\varepsilon]{\varepsilon}_C c' \text{ iff } c = c' \quad \text{and} \quad c \xrightarrow[v \cdot a]{w \cdot b}_C c' \text{ iff } \exists c'' : c \xrightarrow[v]{w}_C c'' \wedge c'' \xrightarrow[a]{b}_C c' \ ,$$

where $v \in (A_C \cup \{0\})^*$ and $w \in B_C^*$ are words of equal length and $a \in A_C$, $b \in B_C$. Moreover, the word resulting from removing all occurrences of 0 from a word $v \in A \cup \{0\}$ is denoted by $\hat{v}$. Now, the interface language of C is $\mathsf{IL}(C) = \{\hat{v} \mid \exists w, c : c_0 \xrightarrow[v]{w}_C c\}$. It is easy to see that $\mathsf{IL}(C)$ is prefix-closed. We say that C *respects* a language $L \subseteq A^*$ if $\mathsf{IL}(C) \subseteq L$.

3 Process Specifications and the Refinement Preorder

We may now formalize the setup indicated in the introduction. An *interface* for a component (=process) P is simply a prefix-closed language $L \subseteq A_P^*$, and describes the protocol agreed upon for using P. Then a new component (=process) Q is a correct upgrade of an old component P that is equipped with interface language L, if it behaves as P in connection with any context that respects L, i.e. if it is drawn from the following set of processes:

$$\mathcal{U}_{P,L} \overset{\text{def}}{=} \{Q \in \mathsf{Proc}_A \mid \forall C \in \mathsf{Ctx}_A : \mathsf{IL}(C) \subseteq L \Rightarrow C(P) \sim C(Q)\} \ .$$

Note that we use strong bisimulation as notion of global behavioral coincidence. $\mathcal{U}_{P,L}$ is thus the set of correct *upgrades of P w.r.t. L*.

As mentioned, interface languages are quite useful for module users, since they provide information they can check their systems against, but less useful to implementors, who would likely prefer a representation that more explicitly states the allowable implementation choices for upgrades. The remainder of this paper is devoted to showing that any set of upgrades can be characterized by a single element in a certain space of simple behavioral *process specifications* that extends the space of processes and is equipped with a behavioral, bisimulation-like refinement preorder.

Definition 3 (Process specifications). *A* process specification *is a pair* $\mathcal{P} = (P, \uparrow)$ *consisting of a process* P *and an* undefinedness predicate $\uparrow \subseteq S_P \times A_P$.

In the following we write $p \uparrow a$ in lieu of $(p, a) \in \uparrow$ and call this an a-undefinedness of state p. Moreover, we write $p \downarrow a$ as an abbreviation for $\neg(p \uparrow a)$. The intuitive interpretation of an a-undefinedness of a state p is that a-transitions from state p are completely irrelevant and might thus arbitrarily be removed or added. This intuition is formally captured by the notion of the refinement preorder defined below.

We use calligraphic letters $\mathcal{P}, \mathcal{Q}$ to denote process specifications. The corresponding italic letters P, Q refer to the embodied processes, and we continue to refer to their constituting parts by $S_P, A_P, \ldots$, $(S_Q, A_Q, \ldots)$ and extend this convention by referring to the undefinedness predicates by $\uparrow_P$ and $\uparrow_Q$.

A process specification $\mathcal{P} = (P, \uparrow)$ is called *total* if no reachable state p has an undefinednesses, i.e. if $p \downarrow a$ for all $a \in A_P$ and reachable $p \in S_P$. Total process specifications correspond to processes, and henceforth we will identify a process P with the corresponding total process specification $(P, \emptyset)$. The definition of the application of contexts to processes extends in a natural way to specifications.

Definition 4 (Application of contexts to specifications). *Suppose* $\mathcal{P} = (P, \uparrow_P)$ *is a process specification and* C *is a context such that* $A_P = A_C$. $C(\mathcal{P})$ *is the process specification* $(C(P), \uparrow)$*, where the undefinedness predicate* $\uparrow$ *is the smallest predicate obeying the rule:*

$$\frac{p \uparrow_P a \;,\; c \xrightarrow[a]{b}_C c'}{c(p) \uparrow b}$$

Note that for a process P, $C((P, \emptyset))$ equals $(C(P), \emptyset)$, the total specification corresponding to $C(P)$.

Definition 5 (Refinement preorder). *Suppose* $\mathcal{P} = (P, \uparrow_P)$ *and* $\mathcal{Q} = (Q, \uparrow_Q)$ *are process specifications with the same alphabet* ($A_P = A_Q$). *A relation* $R \subseteq S_P \times S_Q$ *is called a* pre-bisimulation *if for all* $(p, q) \in R$ *and* $a \in A_P$ *with* $p \downarrow_P a$ *the following three conditions hold:*

- $q \downarrow_Q a$,
- $\forall p' : p \xrightarrow{a}_P p' \Rightarrow \exists q' : q \xrightarrow{a}_Q q' \wedge (p', q') \in R$, *and*
- $\forall q' : q \xrightarrow{a}_Q q' \Rightarrow \exists p' : p \xrightarrow{a}_P p' \wedge (p', q') \in R$.

Define $\preceq$ *to be the union of all pre-bisimulations* R. *Given process specifications* $\mathcal{P}$ *and* $\mathcal{Q}$, *we write* $\mathcal{P} \preceq \mathcal{Q}$ *if* $p_0 \preceq q_0$. *We call* $\mathcal{Q}$ *a* refinement *of* $\mathcal{P}$ *in this case.*

Standard arguments establish that $\preceq$ is itself a pre-bisimulation (viz. the largest one) and a preorder (i.e. a reflexive and transitive relation) on process specifications. We call it the *refinement preorder* and denote its kernel $\preceq \cap \succeq$ by $\asymp$. Moreover, it is easy to see that the notions of bisimulation and pre-bisimulation coincide for processes (i.e. total specifications) because processes possess no undefinednesses.

Contexts are monotonic w.r.t. the refinement preorder, i.e. $P \preceq Q$ implies $C(P) \preceq C(Q)$ for process specifications $\mathcal{P}, \mathcal{Q}$ and contexts C with $A_P = A_Q = A_C$. This monotonicity result generalizes the compositionality of contexts w.r.t. strong bisimulation, as $\sim$ and $\preceq$ coincide for total process specifications.

A process specification $\mathcal{P}$ can be interpreted as representing the set $\mathcal{I}(\mathcal{P}) \overset{\text{def}}{=} \{Q \in \mathsf{Proc}_A \mid \mathcal{P} \preceq Q\}$ of all refining processes, its *implementations.* Note that $\mathcal{I}(\mathcal{P})$ contains only the *total* refinements of $\mathcal{P}$, i.e. processes, not (proper) process specifications. Transitivity of $\preceq$ implies that smaller processes have more implementations, i.e., $\mathcal{P} \preceq \mathcal{Q}$ implies $\mathcal{I}(\mathcal{P}) \supseteq \mathcal{I}(\mathcal{Q})$.

4 Characterizing Replacement Sets by Process Specifications

Given a process P and an interface language L, we would like to construct a process specification $\mathcal{S}_{P,L}$, the upgrade specification promised in the introduction. Clearly, we expect that all implementations of $\mathcal{S}_{P,L}$ can replace P in any context C that respects L. This requirement, which we call $\mathcal{S}_{P,L}$'s *soundness* in the following, can be expressed by the following inclusion:

$$\textit{Soundness} \qquad\qquad \mathcal{I}(\mathcal{S}_{P,L}) \subseteq \mathcal{U}_{P,L} \enspace .$$

Preferably, $\mathcal{S}_{P,L}$ should be as small as possible w.r.t. $\preceq$ in order to characterize as many valid replacements for P as possible. Ideally, we would like that it even characterizes *all* valid replacements for P, which can be expressed by

$$\textit{Completeness} \qquad\qquad \mathcal{I}(\mathcal{S}_{P,L}) \supseteq \mathcal{U}_{P,L} \enspace .$$

We call this property $\mathcal{S}_{P,L}$'s *completeness.* We will see that we can indeed construct $\mathcal{S}_{P,L}$ so that it is both sound and complete.

4.1 Construction of Upgrade Specifications

As L is assumed to be prefix-closed, there is a deterministic process, Q, whose language equals L. One possible construction of such a process Q is the following: as states we take languages over A, i.e. $S_Q \subseteq 2^{A^*}$, the alphabet of Q is $A_Q = A$, the transition relation is defined by the rule

$$\frac{\{w \mid a \cdot w \in q\} \neq \emptyset}{q \xrightarrow{a}_Q \{w \mid a \cdot w \in q\}}$$

and the initial state is $q_0 = L$. More precisely, we restrict S_Q to the languages reachable via $\rightarrow_Q$-transitions from $q_0 = L$. If L is regular, Q corresponds to the minimal deterministic automaton detecting L and, therefore, it is intuitive to call Q also in general the *minimal deterministic process* for the language L.

The observation underlying the construction of $\mathcal{S}_{P,L}$ now is the following: if a component P and a process Q as above run in parallel in a synchronous fashion inside a context C respecting L, then the fact that Q has no a-transition in a certain state means that C cannot consume an a-action in the next step (since it respects L). Therefore, in such a state addition or removal of a-transitions does not change the behavior visible to the environment.

This suggest the following definition: $\mathcal{S}_{P,L} = ((S, A, \rightarrow, (p_0, q_0)), \uparrow)$, where $S = S_P \times S_Q$ (Q is the minimal deterministic process for L from above) and $\rightarrow \subseteq S \times A \times S$ and $\uparrow \subseteq S \times A$ are the smallest relations obeying the rules

$$\frac{p \xrightarrow{a}_P p' \;,\; q \xrightarrow{a}_Q q'}{(p,q) \xrightarrow{a} (p',q')} \qquad \text{and} \qquad \frac{\neg\exists q' : q \xrightarrow{a}_Q q'}{(p,q) \uparrow a} \; .$$

In place of Q we could use any deterministic process with language L. We refer to the *minimal* deterministic process here only for purpose of unique definition.

As an example, we present in Fig. 1(d) the upgrade specification $\mathcal{S}_{P,L}$, where P is the one-place buffer from Fig. 1(a) and $L = (in.out)^*.(in + \varepsilon)$ is its interface language as discussed in the introduction. Note that in this case the minimal deterministic process for L just looks like P itself. Note also that both the original as well as the upgraded buffer from Fig. 1 are refinements of the upgrade specification $\mathcal{S}_{P,L}$.

We clearly expect that P implements $\mathcal{S}_{P,L}$.

Proposition 6. *$\mathcal{S}_{P,L} \preceq P$ for all $P \in \mathsf{Proc}_A$ and prefix-closed $L \subseteq A^*$.*

An intuitive proof of this proposition is that $\mathcal{S}_{P,L}$ can be thought to be constructed from P by the following three transformations, the first and third of which obviously preserve $\asymp$ and the second of which leads to a process that is weaker w.r.t. the refinement preorder $\preceq$:

1. P is unrolled appropriately;
2. a-undefinednesses are added at the states of the unrolled transition system for which further a-evolution is prohibited by L;
3. all a-transitions are removed from states that now contain an a-undefinedness.

4.2 Soundness of Upgrade Specifications

From the intuition underlying the construction of $\mathcal{S}_{P,L}$ it is clear that a context C respecting L will never try to exchange an action a with $\mathcal{S}_{P,L}$ for which $\mathcal{S}_{P,L}$ is undefined. The following lemma intuitively is a consequence of this fact.

Lemma 7. *If $\mathsf{IL}(C) \subseteq L$, then $C(\mathcal{S}_{P,L})$ is total.*

Proof. Suppose $\mathsf{IL}(C) \subseteq L$. We have to show that no reachable state in $C(\mathcal{S}_{P,L})$ has an undefinedness. Let $\mathsf{IL}(c) = \{\hat{v} \mid \exists w, c' : c \xrightarrow[v]{w}_C c'\}$ be the interface language of a state $c \in S_C$.

Suppose that Q is the minimal deterministic process for L used in the construction of $\mathcal{S}_{P,L}$. The states of $C(\mathcal{S}_{P,L})$ have the form $c(p,q)$, where $c \in S_C$, $p \in S_P$, and $q \in S_Q$. Consider the set $G = \{c(p,q) \mid \mathsf{IL}(c) \subseteq L(q)\}$. It is easy to show the following three properties of G:

a) G contains the initial state $c_0(p_0, q_0)$ of $C(\mathcal{S}_{P,L})$.
b) G is closed under transitions of $C(\mathcal{S}_{P,L})$.
c) No state in G has an undefinedness.

These properties suffice to prove the lemma: a) and b) together imply that G contains all reachable states of $C(\mathcal{S}_{P,L})$; c) then yields that no reachable state has an undefinedness. □

It is now easy to show that a process P can be replaced in a context respecting L by any implementation of $\mathcal{S}_{P,L}$.

Theorem 8 (Single contexts). *Suppose $C \in \mathsf{Ctx}_A$ is a context respecting L and $P, Q \in \mathsf{Proc}_A$ are processes. Then $\mathcal{S}_{P,L} \preceq Q$ implies $C(P) \sim C(Q)$.*

Proof. Suppose $\mathcal{S}_{P,L} \preceq Q$. As contexts are monotonic w.r.t. $\preceq$, we can infer that $C(\mathcal{S}_{P,L}) \preceq C(Q)$. By Lemma 7, $C(\mathcal{S}_{P,L})$ is total. As $\preceq$ and $\sim$ coincide for total processes we thus have $C(\mathcal{S}_{P,L}) \sim C(Q)$. In the same way we can infer $C(\mathcal{S}_{P,L}) \sim C(P)$ because $\mathcal{S}_{P,L} \preceq P$ (Proposition 6). We obtain thus $C(P) \sim C(Q)$ as $\sim$ is an equivalence. □

The claim of the above theorem might be called 'soundness of $\mathcal{S}_{P,L}$ for single contexts'. As a corollary, we obtain the soundness of $\mathcal{S}_{P,L}$.

Corollary 9 (Soundness). $\mathcal{I}(\mathcal{S}_{P,L}) \subseteq \mathcal{U}_{P,L}$.

Proof. Suppose that $Q \in \mathcal{I}(\mathcal{S}_{P,L})$. By definition of $\mathcal{I}(\mathcal{S}_{P,L})$, $\mathcal{S}_{P,L} \preceq Q$. By Theorem 8 we have for any of the contexts $C \in \mathsf{Ctx}_A$ considered in the definition of $\mathcal{U}_{P,L}$, $C(P) \sim C(Q)$. Thus $Q \in \mathcal{U}_{P,L}$. □

4.3 Completeness of Upgrade Specifications

The proof of $\mathcal{S}_{P,L}$'s completeness relies on the converse of the implication

$$C(P) \sim C(Q) \;\Rightarrow\; \mathcal{S}_{P,L} \preceq Q$$

in Theorem 8 for certain contexts. In general, however, this implication, which expresses 'completeness of $\mathcal{S}_{P,L}$ for single contexts' is invalid in the situation of Theorem 8. Note that completeness of $\mathcal{S}_{P,L}$ only means validity of the weaker implication

$$(\forall C : \mathsf{IL}(C) \subseteq L \Rightarrow C(P) \sim C(Q)) \;\Rightarrow\; \mathcal{S}_{P,L} \preceq Q \;.$$

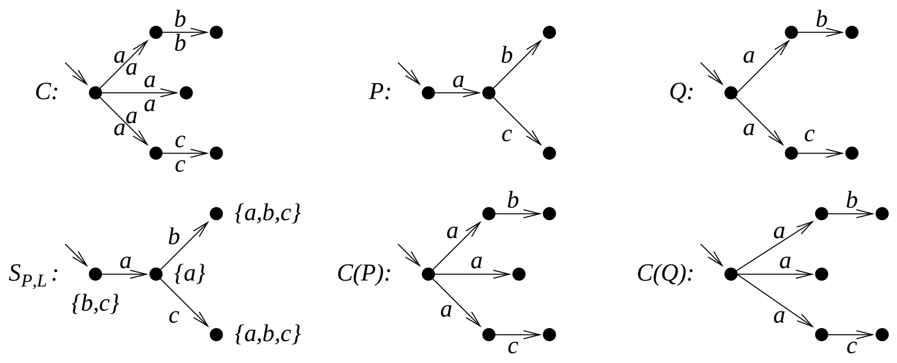

Fig. 2. Incompleteness for non-deterministic context.

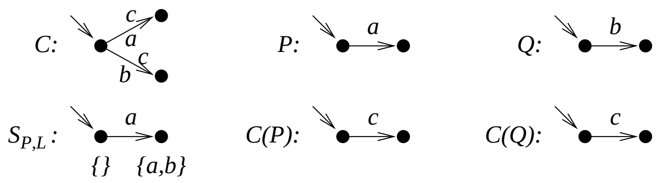

Fig. 3. Incompleteness for non-distinctive context.

The following three phenomena contribute to the incompleteness for single contexts.

Firstly, a context can have different transductions which consume the same action from the parameter process and produce the same action for the environment but lead to different context states. We say then that the context is *non-deterministic.* An example is shown in Fig. 2, where the undefinedness predicate $\uparrow$ of $\mathcal{S}_{P,L}$ is shown by annotating the states with the set of actions for which it is undefined. The alphabets in the example of Fig. 2 are $A = B = \{a, b, c\}$ and $L = \mathsf{IL}(C) = \{\varepsilon, a, ab, ac\}$. Clearly, $C(P) \sim C(Q)$ but $\mathcal{S}_{P,L} \preceq Q$ does not hold. The problem is that due to the non-determinism of the context the branching structure of a parameter process P need not necessarily be preserved in correct replacements Q for that context. Like bisimulation, however, the refinement preorder $\preceq$ preserves branching.

Secondly, a context can exchange different actions with its parameter process and yet produce the same observable behavior. We say that the context is *not distinctive* in this case. A simple example is given in Fig. 3 where the alphabets are $A = \{a, b\}$, $B = \{c\}$ and L is chosen to be $\mathsf{IL}(C) = \{\varepsilon, a, b\}$. Again $C(P) \sim C(Q)$, although $\mathcal{S}_{P,L} \preceq Q$ does not hold. Correct replacements in contexts that are not distinctive cannot (always) completely be described by a process specification because $\preceq$ preserves the identity of actions.

A third, somewhat less severe problem is that C can have a properly smaller interface language than L. A very simple example of this kind is presented in Fig. 4. Here, $A = \{a\}$ and B is arbitrary. We choose $L = A^*$ which certainly is

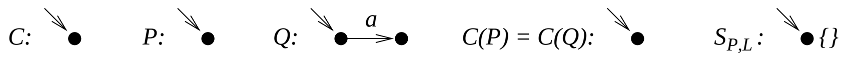

Fig. 4. Incompleteness for context with a strictly smaller interface language.

a superset of $\mathsf{IL}(C) = \emptyset$. Trivially, $C(P) \sim C(Q)$ but $\mathcal{S}_{P,L} \preceq Q$ is invalid. The problem is that L requires the preservation of more from the behavior of the parameter process P than necessary for the context C.

Before we proceed, let us define the notion of deterministic and distinctive contexts referred to above.

Definition 10 (Deterministic and distinctive contexts). *A context* $C = (S, A, B, \rightarrow, s)$ *is called* deterministic *if for all* $c, d, d' \in S$, $a \in A \cup \{0\}$, $b, b' \in B$ *the implication*

$$c \xrightarrow[a]{b} d \wedge c \xrightarrow[a]{b'} d' \quad \Rightarrow \quad b = b' \wedge d = d'$$

is valid and, furthermore, $c \xrightarrow[a]{b} d \wedge c \xrightarrow[0]{b'} d' \quad \Rightarrow \quad a = 0$.

It is called distinctive *if for all* $c, d, d' \in S$, $a, a' \in A$, $b \in B$ *the following implication holds:*

$$c \xrightarrow[a]{b} d \wedge c \xrightarrow[a']{b} d' \wedge (\exists P, P' \in \mathsf{Proc}_A : d(P) \sim d'(P')) \quad \Rightarrow \quad a = a' \ .$$

Here we identify the context states d *and* d' *with the contexts* D *and* D' *that possess the same components as* C *except of the start states which are* d *and* d' *respectively.*

Determinacy is intended to capture the idea of unique transduction: the external effect induced by an action a consumed from the parameter process is required to be uniquely determined. Autonomous context steps might in general prohibit to transfer this property of unique transduction required in the first condition for *single* actions to whole consumed *action sequences*. Therefore, the second condition requires in addition that 0-transductions (which are unique by the first condition) do not compete with non-0 transductions.

While determinacy allows the inference of outer behavior from inner behavior, the idea of distinctivity is to allow just the opposite: to infer inner behavior form outer behavior. Let us, for the purpose of explanation, look first at a somewhat simpler notion, *local distinctivity*, that requires the stronger implication

$$c \xrightarrow[a]{b} d \wedge c \xrightarrow[a']{b} d' \quad \Rightarrow \quad a = a' \ .$$

A locally distinctive context allows to infer from a certain action b observed by the environment immediately the action a of the component process inducing b. The weaker notion of distinctivity does not necessarily allow *immediate* inference of a from the observed action b alone but from b together with the future behavior. Thus detection of a might be delayed but is conceptually possible from the total behavior presented to the environment.

The following lemma shows that the above list of phenomena leading to incompleteness of $\mathcal{S}_{P,L}$ for single contexts is comprehensive.

Theorem 11 (Completeness for single contexts). *Suppose $C \in \mathsf{Ctx}_A$ is a context, $L \subseteq A^*$ is a prefix-closed language, and $P, Q \in \mathsf{Proc}_A$ are processes. If C is deterministic and distinctive and $\mathsf{IL}(C) = L$ then $C(P) \sim C(Q)$ implies $\mathcal{S}_{P,L} \preceq Q$.*

Proof. Suppose that C is deterministic and distinctive, that $\mathsf{IL}(C) = L$, and that $C(P) \sim C(Q)$. Let R be the minimal deterministic process for L used in the construction of $\mathcal{S}_{P,L}$. Given the determinacy and distinctivity of C it is rather straightforward (albeit tedious) to show that the relation

$$S \overset{\text{def}}{=} \{((p,r),q) \mid \exists c : \mathsf{IL}(c) = L(r) \text{ and } c(p) \sim c(q)\}$$

is a pre-bisimulation between $\mathcal{S}_{P,L}$ and Q. This establishes the claim of the theorem, as $((p_0, r_0), q_0)$, the pair of initial states of $\mathcal{S}_{P,L}$ and Q, is contained in R because $\mathsf{IL}(c_0) = \mathsf{IL}(C) = L = L(R) = L(r_0)$ and $C(P) \sim C(Q)$. □

Theorem 11 shows that deterministic distinctive contexts that exhaust the agreed protocol language L (i.e. $\mathsf{IL}(C) = L$) are of particular importance. We call such contexts *witness contexts for L*. That witness contexts always exist is the claim of the following lemma. As a consequence $\mathcal{S}_{P,L}$ is complete.

Lemma 12. *There is a witness context for any prefix-closed language L.*

Proof. Suppose given a prefix-closed language $L \subseteq A^*$. Let – as in the construction of $\mathcal{S}_{P,L}$ – $Q = (S_Q, A, \rightarrow_Q, q_0)$ be the minimal deterministic process for L. Consider the context $C = (S_Q, A, A, \rightarrow_C, q_0)$, where $\rightarrow_C$ is defined by $c \xrightarrow[a]{b}_C c'$ iff $c \xrightarrow{a}_Q c' \wedge a = b$. It is straightforward to check that C is deterministic and distinctive and that its interface language equals $L(Q) = L$. □

Corollary 13 (Completeness). *$\mathcal{U}_{P,L} \subseteq \mathcal{I}(\mathcal{S}_{P,L})$ for any prefix-closed $L \subseteq A^*$.*

Proof. Suppose given $Q \in \mathcal{U}_{P,L}$. By Lemma 12 there is a witness context C for L. As $Q \in \mathcal{U}_{P,L}$ we have $C(P) \sim C(Q)$. By Lemma 11, therefore, $\mathcal{S}_{P,L} \preceq Q$, i.e. $Q \in \mathcal{I}(\mathcal{S}_{P,L})$. □

5 Restricted Context Classes

The results of the previous section seem to depend on the richness of the space of contexts given by action transducers. While this richness certainly is welcome for soundness considerations – it means that replacement in any reasonable context is correct – it is less clear whether it should also be accepted for completeness considerations: reasonable smaller classes of contexts, which could result e.g. from syntactic restrictions on the way contexts are constructed, might allow more replacements, thereby rendering $\mathcal{S}_{P,L}$ incomplete.

Assume that we are interested in a certain context class $K \subseteq \mathsf{Ctx}_A$. Then the set of correct upgrades for a certain process $P \in \mathsf{Proc}_A$ for contexts in K respecting a certain interface language $L \subseteq A^*$ is given by

$$\mathcal{U}^K_{P,L} \overset{\text{def}}{=} \{Q \in \mathsf{Proc}_A \mid \forall C \in K : \mathsf{IL}(C) \subseteq L \Rightarrow C(P) \sim C(Q)\} .$$

The only difference to the definition of $\mathcal{U}_{P,L}$ is the relativation of the universal quantifier to contexts in K. It is obvious that $\mathcal{U}_{P,L} \subseteq \mathcal{U}^K_{P,L}$ because $K \subseteq \mathsf{Ctx}_A$. Therefore, by Corollary 9, $\mathcal{I}(\mathcal{S}_{P,L}) \subseteq \mathcal{U}^K_{P,L}$, i.e. $\mathcal{S}_{P,L}$ is *sound for* K. The more interesting question is, whether it is also *complete for* K, i.e. whether $\mathcal{I}(\mathcal{S}_{P,L}) \supseteq \mathcal{U}^K_{P,L}$. In order to show completeness, however, it suffices to find a witness context $C \in K$ for L because in this case we can argue as in the proof of Corollary 13. Summarizing we have the following.

Corollary 14. *Suppose $L \subseteq A^*$ is prefix-closed. If K contains a witness context for L, then $\mathcal{S}_{P,L}$ is sound and complete for K, i.e. $\mathcal{I}(\mathcal{S}_{P,L}) = \mathcal{U}^K_{P,L}$.*

In the scenario motivating the considerations of this paper the process library components to be replaced typically run in parallel with the using program. Therefore, parallel contexts are of particular interest. Two prominent views of parallel interaction studied in the realm of process algebra are multiple agreement as in CSP [4] or LOTOS and handshake communication as in CCS [9].

In CSP the parallel composition operator *enforces* synchronization between its components on the common parts of their alphabet. Therefore, in a parallel CSP context of the form $\cdot \parallel Q$ the process Q can control occurence of actions in components placed into such a context. Technically this means that parallel CSP contexts have in general a non-trivial interface language and are thus interesting from the point of view of this paper.

The parallel composition operator of CCS, on the other hand, does not enforce synchronization of the component processes but only *enables* it. In particular, the component processes of a pure parallel composition can proceed independently of each other and, therefore, the interface language of a pure parallel CCS context $\cdot|Q$ (for processes of alphabet A) is just A^*. Thus pure parallel CCS context are of little interest from the point of view of this paper. A more interesting context class, which subsumes parallel contexts, are *standard concurrent contexts* of the form $(\cdot|Q) \setminus M$. Here, the *restriction operator* $\cdot \setminus M$ of CCS is used to enforce synchronization on the action set $M \subseteq \mathsf{Act}$. Standard concurrent contexts are modeled on processes in *standard concurrent form*[1] that are often studied in the realm of CCS (see, e.g., [9]).

In the full version of this paper we recall how to capture the effect of parallel CSP contexts and standard concurrent CCS contexts by action transducers and demonstrate that both context classes contain witness contexts for given interface languages L. Thus, $\mathcal{S}_{P,L}$ is sound and complete for each of these classes.

In the CSP case, witness contexts are rather immediately induced by deterministic processes for the language L in question with a certain care for treating the internal action τ correctly. In contrast the CCS case faces us with a difficulty: standard concurrent contexts straightforwardly constructed from such processes are non-distinctive in general. The reason is the implicit hiding of the

[1] A CCS process is said to be in standard concurrent form if it has the form $(P_1[f_1]|\ldots|P_n[f_n]) \setminus M$. For the purpose of this paper Q can be thought to represent the parallel composition $P_2[f_2]|\ldots|P_n[f_n]$ and the relabeling $[f_1]$ can be thought to be subsumed by the component placed into the context.

CCS parallel operator: a synchronization of two complementary actions a and $\bar{a}$ yields just the internal τ-action from which we cannot infer which actions synchronized. (Note that this is different in CSP where actions are not changed on synchronization.) How can we nevertheless construct distinctive contexts? The idea is to enrich the context to output *tracing actions* after each synchronization, from which the actions that synchronized can be inferred. This is akin to including some special output statements into a program for debugging purposes in order to observe the path taken through the program. The resulting context is distinctive in the sense of Definition 10, although it is not locally distinctive. Indeed, this observation was the reason to opt for the more global notion of distinctivity.

6 Conclusion

The motivation for this paper is to initiate a theory of downwards compatibility for reactive components. To this end, we studied the use of interface languages as a means for constraining the applications in which a reactive component is to be used. Such languages describe admissible interaction patterns of applications. As the main technical result we showed how to construct library-developer-oriented upgrade specifications from components equipped with interface languages.

While suggestive, the results in this paper represent only a first step toward an adequate theory of component compatibility. In particular, we deliberately ignored value-passing in order to come to grips with the control-oriented aspects of reactive systems. It would be an interesting topic for future research to extend the framework to a more realistic scenario where, in particular, values are communicated between components and applications and to consider how the results of this paper can be applied and generalized. We anticipate that interface languages would then consist of sequences of actions annotated with types, with one such sequence representing a set of admissible sequences of value exchanges. The consequences of this change remain to be investigated. Another topic to be investigated would involve the consideration of more reasonable notions of global behavioral equivalence, in particular weak bisimulation. Results in [2] suggest that this extension is not problematic, so in this paper we have opted for the simpler, if less realistic, setting of strong bisimulation.

From a more technical point of view, this paper has presented a behavioral refinement preorder on a space of simple process specifications and has shown that the correct replacements of a process in context classes given by prefix-closed interface languages can be characterized by single specifications. This expressiveness result for the space of process specifications draws its inspiration from a similar result for classes of CCS contexts in [2]. That paper proposes a modification in the definition of the CCS *divergence preorder* studied by Walker [11]; the modification enables such a result to hold.

In this paper we simultaneously extended and simplified the underlying formalism of [2] to obtain more general soundness and completeness results, and we showed how they may be applied in the setting of reactive component evolution. Specifically, we considered the more general setting of action transducer contexts proposed by Larsen and Xinxin [7, 8]; we altered the setting of [2] to account for this richer setting; and we showed how components together with "interaction languages" may be transformed into equivalent "partial process" specifications. Moreover, we showed that the complete characterization property of the resulting specifications is stable under reasonable modifications of the type of considered contexts. In particular, we studied, besides the comprehensive class of action transducer contexts, the less extensive classes of parallel CSP contexts and standard concurrent CCS contexts.

The refinement preorder has a bisimulation-like definition and can – for finite-state processes – automatically be checked by adapting known bisimulation-checkers. The finiteness requirement imposed by straightforward automatic support, however, leads also to a restriction to *regular* interface languages, as non-regular languages would give rise to infinite process specifications $\mathcal{S}_{P,L}$.

Related to the characterization of correct replacements in contexts is the context decomposition problem studied by Larsen [6, 7]. His work is concerned with characterizing the class of processes Q such that $C(Q)$ sat S holds for a given specification S and context C. Indeed, characterizing the correct replacements of a process P in a *single* context C can be seen as the context decomposition problem, where specifications are given by processes, sat is chosen as the global process equivalence $\sim$ and S as $C(P)$. Context decomposition amounts then to characterizing the processes Q with $C(Q) \sim C(P)$, i.e. the correct replacements of P. The replacement problem in *classes* of contexts, however, that was considered in this paper does not immediately reduce to a context decomposition problem, and our results therefore are fundamentally different from Larsen's.

References

1. J. C. M. Baeten and W. P. Weijland. *Process Algebra.* Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1990.
2. R. Cleaveland and B. Steffen. A preorder for partial process specification. In *CONCUR'90*, LNCS 458, 141–151. Springer-Verlag, 1990.
3. M. C. Hennessy. *Algebraic Theory of Processes.* MIT Press, 1988.
4. C. A. R. Hoare. *Communicating Sequential Processes.* Prentice Hall, 1985.
5. C. B. Jones. Tentative steps toward a development method for interfering programs. *ACM TOPLAS*, 5(4):596–619, 1983.
6. K. G. Larsen. *Context-Dependent Bisimulation Between Processes.* PhD thesis, University of Edinburgh, 1986.
7. K. G. Larsen. Ideal specification formalism = expressivity + compositionality + decidability + testablity + $\cdots$. In *CONCUR'90*, LNCS 458. Springer-Verlag, 1990.
8. K. G. Larsen and L. Xinxin. Compositionality through an operational semantics of contexts. In *ICALP'90*, LNCS 443, 526–539. Springer-Verlag, 1990.
9. R. Milner. *Communication and Concurrency.* Prentice Hall, 1989.
10. D. M. R. Park. Concurrency and automata on infinite sequences. In LNCS 154, pages 561–572. Springer-Verlag, 1981.
11. D. J. Walker. Bisimulations and divergence. In *LICS'88*, 186–192. IEEE Computer Society, 1988.