Compositionality in Model-Based Testing

Gijs van Cuyck[1(✉)], Lars van Arragon[1], and Jan Tretmans[1,2]

[1] Institute iCIS, Radboud University, Nijmegen, The Netherlands
{gijs.vancuyck,lars.vanarragon,jan.tretmans}@ru.nl
[2] TNO-ESI, Eindhoven, The Netherlands

Abstract. Model-based testing (MBT) promises a scalable solution to testing large systems, if a model is available. Creating these models for large systems, however, has proven to be difficult. Composing larger models from smaller ones could solve this, but our current MBT conformance relation **uioco** is not compositional, i.e. correctly tested components, when composed into a system, can still lead to a faulty system. To catch these integration problems, we introduce a new relation over component models called *mutual acceptance*. Mutually accepting components are guaranteed to communicate correctly, which makes MBT compositional. In addition to providing compositionality, mutual acceptance has benefits when retesting systems with updated components, and when diagnosing systems consisting of components.

Keywords: model-based testing · component-based testing · compositional testing · labelled transition systems · uioco

1 Introduction

Modern software systems are becoming increasingly large and complex. Traditional testing scales poorly for systems of these sizes. This causes the development and maintenance of test suites to become costly and time consuming, which slows down the development of new functionality. Model-Based Testing (MBT) is a technique that has been developed to increase the efficiency and effectiveness of testing. With MBT, testers create a model of the system under test from which an MBT tool can then automatically generate and execute test cases. This reduces the problem of creating and maintaining a test suite to creating and maintaining a model of the system under test.

Creating models for complex systems, however, is still difficult and laborious, since often no single person understands the whole system well enough. A solution is to divide and conquer: the system is decomposed into its components which are modelled and tested separately. This requires that the applied MBT methodology is *compositional*: if each component implementation is correct with

This work is part of the project *TiCToC - Testing in Times of Continuous Change*, project nr 17936, part of the research program *MasCot - Mastering Complexity*, which is supported by the Dutch Research Council NWO.

Published by Springer Nature Switzerland AG 2023
S. Bonfanti et al. (Eds.): ICTSS 2023, LNCS 14131, pp. 202–218, 2023.
https://doi.org/10.1007/978-3-031-43240-8_13

respect to its component model, then it can be inferred that the composition of component implementations, i.e. the system under test, is correct with respect to the composition of component models, i.e. the system model.

In this paper, we investigate compositionality for MBT with labelled transition systems as models, **uioco** as the conformance relation, and parallelism modelling component composition [4]. We define a relation over component specification models, called *mutual acceptance*, which guarantees that components communicate neatly, and that **uioco** is preserved under composition. We generalise existing results on compositionality [4,8,11] by making less restrictive assumptions and using a composition operator that is associative so that also compositions of more than two components can be easily considered. Moreover, we use the more recent **uioco** conformance relation instead of **ioco** [19]. A more detailed comparison with related work can be found in Sect. 8.

In addition to compositionality, mutual acceptance also benefits testing evolving systems and software product lines. It enables more effective testing when a component is replaced by an updated version, as will be elaborated in Sect. 7. Diagnosis is the converse of compositionality: if the whole system has a failure, then diagnosis tries to localise the failure in one of its components; Sect. 7 will also discuss the use of mutual acceptance in diagnosis.

Overview. Section 2 contains preliminaries. Section 3 shows why the current approach to compositional model-based testing is not desirable by means of an example. Section 4 formalises what it means for two models to be compatible with each other for use in model-based testing, and defines the mutual acceptance relation $\leftrightharpoons$. Then Sect. 5 goes on to prove that this leads to desirable properties, after which Sect. 6 revisits the example. Section 7 discusses how these properties also lead to a reduced testing effort when substituting components, and how $\leftrightharpoons$ can be used in diagnosis. Section 8 describes some of the large body of related work previously done in the area of compositional model-based testing. Finally, Sects. 9 and 10 discuss possible future work and summarise the main results of this paper, respectively. All proofs for lemmas and theorems can be found in the extended version of this paper [6].

2 Preliminaries

We give the formal definitions for the MBT theory that we consider. We base our work on the theory developed in [4,18]. The main formalism used is that of labelled transition systems (LTS) (Definition 1). An LTS has states and transitions between states that model events. An event can be an input, an output or τ; τ represents an internal transition which is not observable from the outside and can therefore not be tested. I_s, U_s, etc., indicate inputs and outputs, respectively, coming from LTS s. The shorthand L_s means $I_s \cup U_s$. The name of an LTS is sometimes used as shorthand for its starting state. $\mathcal{LTS}(I, U)$ denotes the domain of labelled transition systems with inputs I and outputs U, or just $\mathcal{LTS}$ if I and U are known. For technical reasons we restrict this class to strongly converging and image-finite systems. Strong convergence means that infinite

sequences of τ-actions are not allowed to occur. Image-finiteness means that the number of non-deterministically reachable states shall be finite. In examples, inputs and outputs are given implicitly by prefixing inputs with ?, and outputs with !. The same label can be in the input set of one LTS and in the output set of another.

Definition 1. *A* Labelled Transition System *is a 5-tuple* $\langle Q, I, U, T, q_0 \rangle$ *where:*

- Q *is a non-empty, countable set of states;*
- I *is a countable set of input labels;*
- U *is a countable set of output labels, which is disjoint from* I*;*
- $T \subseteq Q \times (I \cup U \cup \{\tau\}) \times Q$ *is a set of triples, the transition relation;*
- $q_0 \in Q$ *is the initial state.*

Reasoning about labelled transition systems uses the concept of traces. A trace is a sequence of labels that can occur when walking trough an LTS. Common notation used when describing traces is repeated in Definition 2.

Definition 2. *Let* $s \in \mathcal{LTS}$*;* $p_1, p_2 \in Q_s$*;* $\ell \in L_s$*;* $\sigma \in L_s^*$*;* $\ell_\tau \in L_s \cup \{\tau\}$*;* $\sigma_\tau \in (L_s \cup \{\tau\})^*$*, where* ϵ *denotes the empty sequence of labels.*

$$
\begin{aligned}
p_1 \xrightarrow{\epsilon} p_2 &\stackrel{def}{=} p_1 = p_2 \\
p_1 \xrightarrow{\ell_\tau} p_2 &\stackrel{def}{=} (p_1, \ell_\tau, p_2) \in T_s \\
p_1 \xrightarrow{\ell_\tau \cdot \sigma_\tau} p_2 &\stackrel{def}{=} \exists p_3 \in Q_s : p_1 \xrightarrow{\ell_\tau} p_3 \wedge p_3 \xrightarrow{\sigma_\tau} p_2 \\
p_1 \xrightarrow{\sigma} &\stackrel{def}{=} \exists p_3 \in Q_s : p_1 \xrightarrow{\sigma} p_3 \\
p_1 \not\xrightarrow{\sigma} &\stackrel{def}{=} \nexists p_3 \in Q_s : p_1 \xrightarrow{\sigma} p_3 \\
p_1 \stackrel{\epsilon}{\Rightarrow} p_2 &\stackrel{def}{=} \exists \varphi \in \{\tau\}^* : p_1 \xrightarrow{\varphi} p_2 \\
p_1 \stackrel{\sigma \cdot \ell}{\Longrightarrow} p_2 &\stackrel{def}{=} \exists p_3, p_4 \in Q_s : p_1 \stackrel{\sigma}{\Rightarrow} p_3 \wedge p_3 \xrightarrow{\ell} p_4 \wedge p_4 \stackrel{\epsilon}{\Rightarrow} p_2 \\
p_1 \stackrel{\sigma}{\Rightarrow} &\stackrel{def}{=} \exists p_3 \in Q_s : p_1 \stackrel{\sigma}{\Rightarrow} p_3 \\
p_1 \stackrel{\sigma}{\not\Rightarrow} &\stackrel{def}{=} \nexists p_3 \in Q_s : p_1 \stackrel{\sigma}{\Rightarrow} p_3
\end{aligned}
$$

While specifications are often given as an LTS, $IOTS$ are used to represent implementations. In MBT, we commonly assume that we can always give any input to an implementation. $\mathcal{IOTS}$ denotes the domain of all input-enabled transition systems, and $\mathcal{IOTS}(I, U)$ denotes the domain of all input enabled transition systems with input set I and output set U.

Definition 3. $i \in \mathcal{LTS}$ *is an* Input-Enabled Transition System *(IOTS) if in every state for every input, its transition relation either contains that input, or reaches with just internal transitions another state that does so:*

$$\forall q \in Q_i,\ \ell \in I_i :\ q \stackrel{\ell}{\Rightarrow}$$

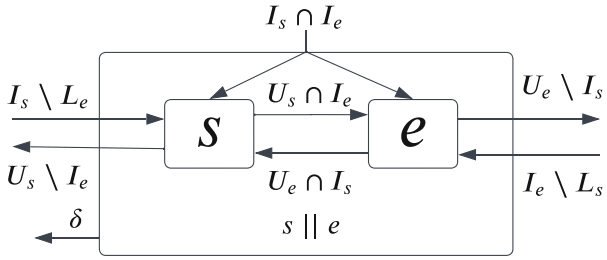


Fig. 1. Parallel composition of system s and its environment e.

Multiple labelled transition systems can be composed to form larger models. For component specifications this is often done using parallel composition (Definition 5). The result of parallel composition represents a system where all the components are being executed at the same time independently of each other. Synchronisation occurs on shared labels. An overview of the label sets of a parallel composition is shown in Fig. 1. Note that we do not require the input sets of the components to be disjoint, which will be explained below. Parallel composition assumes synchronous communication between components. Systems with asynchronous communication can still be modelled, but this requires giving explicit specification for the communication medium.

Definition 4. $s, e \in \mathcal{LTS}$ *are* composable *iff their respective output sets* U_s *and* U_e *are disjoint:* $U_s \cap U_e = \emptyset$

Definition 5. Parallel composition $||$ *on two composable labelled transition systems* s *and* e *is defined as:* $s \,||\, e \stackrel{def}{=} \langle Q, I, U, T, q_0 \rangle$, *where*

- $Q = \{ p \,||\, q \mid p \in Q_s, q \in Q_e \}$
- $I = (I_s \setminus U_e) \cup (I_e \setminus U_s)$
- $U = U_s \cup U_e$
- $q_0 = q_{0s} \,||\, q_{0e}$
- T *is the minimal set satisfying the following inference rules (where* $p, p_1, p_2 \in Q_s, q, q_1, q_2 \in Q_e$*):*

$$\frac{p_1 \xrightarrow{\ell} p_2 \qquad \ell \in (L_s \cup \{\tau\}) \setminus L_e}{p_1 \,||\, q \xrightarrow{\ell} p_2 \,||\, q} \qquad \frac{q_1 \xrightarrow{\ell} q_2 \qquad \ell \in (L_e \cup \{\tau\}) \setminus L_s}{p \,||\, q_1 \xrightarrow{\ell} p \,||\, q_2}$$

$$\frac{p_1 \xrightarrow{\ell} p_2,\ q_1 \xrightarrow{\ell} q_2 \qquad \ell \in L_s \cap L_e}{p_1 \,||\, q_1 \xrightarrow{\ell} p_2 \,||\, q_2}$$

Lemma 1. *Parallel composition is* commutative *and* associative *(up to isomorphism* $\equiv$*), i.e. for* $s, e, t \in \mathcal{LTS}$*, we have:*

$$\begin{array}{lrcl} \textit{commutativity}: & s \,||\, e & \equiv & e \,||\, s \\ \textit{associativity}: & (s \,||\, e) \,||\, t & \equiv & s \,||\, (e \,||\, t) \end{array}$$

Our definition for *composable* is weaker than the one in other papers: $I_s \cap I_e = U_s \cap U_e = \emptyset$ [1,4,7]. This is because requiring disjoint input sets leads to a composition operator that is not associative [3]. A more detailed discussion of the properties of various types of parallel composition can be found in [20]. With our less restrictive definition of *composable*, parallel composition is both associative and commutative as expressed in Lemma 1. This is important, as it means that more than two components can also be composed and the order in which components are composed does not matter. The remaining restriction of disjoint output sets does not really restrict the applicability of parallel composition. Output sets can always be made disjoint by renaming one output label and then duplicating the synchronising transitions for the new label.

Another common approach to parallel composition is to replace all synchronised transitions with τ transitions. This is done under the assumption that communication between components is by default not observable by the outside world and therefore should be hidden. A downside is that this removes information, which makes specification-based analysis less useful. Additionally, a large part of the model-based testing theory assumes convergence, i.e. the absence of divergence. This means that there are no infinite paths of just τ-transitions possible in the specification. By automatically hiding the labels of synchronised transitions, divergence is often introduced into the composed specification. For these reasons, we choose not to automatically hide labels during composition.

The main purpose of a labelled transition system when used for model-based testing is to describe when an implementation is considered correct. This is done through a conformance relation.

Two common conformance relations are **ioco** [18] and the more recent **uioco** relation [4]. **uioco** differs from **ioco** in how it deals with *nondeterministic underspecification*, i.e. how non-specified inputs are handled. Among others, **uioco** is better suited for reasoning about composition. A detailed comparison of the two relations can be found in [19].

Definition 6. *For $s \in \mathcal{LTS}$, $\delta \notin L_s$ is a special output denoting the absence of outputs, called* quiescence. *It is defined as follows (with $p_1, p_2 \in Q_s$):*

$$p_1 \xrightarrow{\delta} p_2 \quad \stackrel{def}{=} \quad p_1 = p_2 \ \wedge \ \forall x \in U_s \cup \{\tau\} : \ p_1 \not\xrightarrow{x}$$

L^δ, U^δ is used as shorthand for $L \cup \{\delta\}$, $U \cup \{\delta\}$ respectively.

Definition 7. *Let $s \in \mathcal{LTS}$; $p_1 \in Q_s$; $P \subseteq Q_s$ and $\sigma \in L_s^{\delta *}$.*

$$\begin{array}{lcl} p_1 \ \textbf{\textit{after}} \ \sigma & \stackrel{def}{=} & \{\ p_2 \in Q_s \ \mid \ p_1 \stackrel{\sigma}{\Rightarrow} p_2 \ \} \\ \textbf{\textit{out}}(p_1) & \stackrel{def}{=} & \{\ x \in U_s^\delta \ \mid \ p_1 \xrightarrow{x} \ \} \\ \textbf{\textit{out}}(P) & \stackrel{def}{=} & \bigcup \{\ \textbf{\textit{out}}(p) \ \mid \ p \in P \ \} \end{array}$$

Definition 8. *Let $i \in \mathcal{IOTS}(I,U)$; $s \in \mathcal{LTS}(I,U)$:*

$$\begin{aligned} \textbf{\textit{Utraces}}(s) &\stackrel{def}{=} \{\ \sigma \in L^{\delta *} \mid s \stackrel{\sigma}{\Rightarrow} \wedge\ (\ \nexists p \in Q_s,\ \sigma_1 \cdot a \cdot \sigma_2 = \sigma : \\ &\qquad\qquad a \in I \ \wedge\ s \stackrel{\sigma_1}{\Rightarrow} p \ \wedge\ p \stackrel{a}{\not\Rightarrow}\ )\ \} \\ i\ \textbf{\textit{uioco}}\ s &\stackrel{def}{=} \forall \sigma \in \textbf{\textit{Utraces}}(s) :\ \textbf{\textit{out}}(i\ \textbf{\textit{after}}\ \sigma) \subseteq \textbf{\textit{out}}(s\ \textbf{\textit{after}}\ \sigma) \end{aligned}$$

3 Motivating Example: A Parking System

We argue that parallel composition does not work nicely with **uioco**, which we will show with an example in this section. Consider two components that together function as an automatic parking system in a car: a sensor which observes the environment and an actuator that parks the car. An illustration of how these two components communicate with each other and their environment is shown in Fig. 2. Specifications for the behaviour of these components are shown in solid black in Fig. 3. Their behaviour is straightforward: the parking component keeps parking as long as the sensor tells it that it is safe to do so, but stops parking if there is an obstacle, at which point it will stop the car and turn the sensor off. These components are left under-specified on purpose: it does not really matter what the sensor does if it detects an obstacle after it has been turned off, as long as it does not start beeping. This gives an implementer of the actual sensor some freedom, but still specifies the important behaviour.

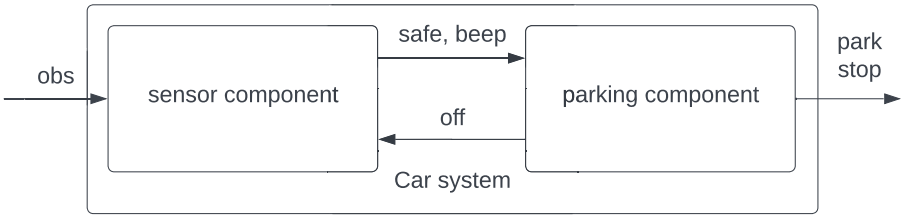
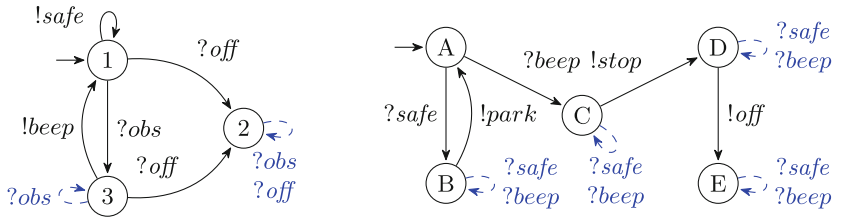


Fig. 2. Two component parking system

Possible implementations that are **uioco** correct are also given in Fig. 3 using the extra dashed blue transitions. On first glance this all seems to make sense, and model-based testing will not find any problems when testing the components. I.E. I_1 **uioco** $S_1 \wedge I_2$ **uioco** S_2. After composing our components using parallel composition, however, which is shown in Fig. 4, the composed implementation is not **uioco** correct to the composed specification.

The problem with the implementation in Fig. 4 is that it contains unspecified output transitions. These can be seen as some of the dashed transitions, which are only present in the implementation and not in the specification. This means that the previously valid implementations are now generating outputs that are not



(a) S_1 and I_1: car sensor component **(b)** S_2 and I_2: automated parking component

Fig. 3. Car component specifications (→) and implementations (- →)

part of the composed specification. Model-based testing will report an error here, while the components are actually behaving as specified. Additionally, hidden within these false positives, there is also an actual error: if the sensor detects an obstacle after already having communicated that there is no obstacle, the parking system will not respond and will just continue parking. This is represented by the *!beep* transition from B3 to B1, which could for instance happen if a moving obstacle like a person is present. This shows that only looking at the individual components is not enough, as there are real problems that only become visible when looking at combinations of components together.

We argue that the main problem with this example is that the component specifications rely on unspecified behaviour. The sensor specification describes exactly when the sensor is allowed to beep, but the parking specification does not always specify what the result should be. There is no guarantee that the result does not crash the system or violate any requirements. One way this could be resolved is by expanding the specifications to be input complete [4]. However, doing so would remove the possibility for under-specification, which is a desirable feature in modelling behaviour. Under-specification keeps models smaller and more readable, and gives more freedom when implementing the specification. Another approach is therefore desired: a specification should specify all the behaviour that is used by other specifications, but leave the possibility of not specifying unused behaviour. This goal will be made more concrete in Sect. 4.

4 Mutual Acceptance

In order to reason about specified and unspecified behaviour an explicit notion of what it means for behaviour to be specified is required. For **uioco**, the allowance of outputs is always explicitly specified. They are either present in the model and therefore allowed, or absent and disallowed. After a specified input, the model again defines what is allowed. Inputs are always implicitly allowed, but if an input is not part of the model all behaviour after that input is allowed. This means that the behaviour after an absent input is unspecified: the model does not tell us what should or should not happen. Therefore, if all outputs given by one component, are inputs present in the model of the other component, there will be no unspecified behaviour. This requirement is formulated in Definitions 9

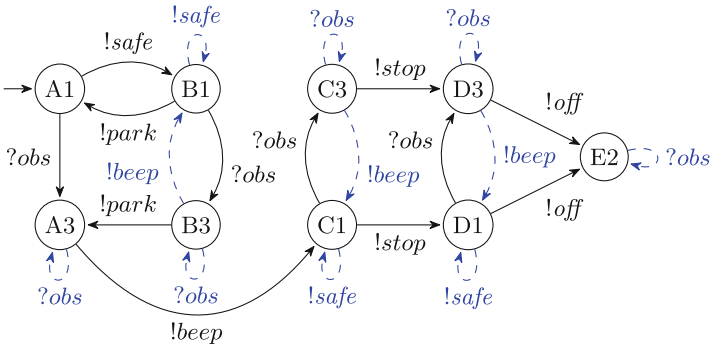


Fig. 4. Car autopark and sensor composed $S_1||S_2$ (→) and $I_1||I_2$ (⇢)

to 11: if after some $\sigma \in \mathbf{Utraces}(s \,||\, e)$ some pair of states s', e' is reached, and s' produces a synchronised output, then e' must have this output as an input. Note that this is trivially holds if e is input enabled which generalises earlier results about component-based testing with **uioco** [4].

Definition 9. *For $s \in LTS$; $p \in Q_s$; $P \subseteq Q_s$, the set of enabled inputs is defined as:*

$$\begin{aligned} \boldsymbol{in}(p) &\stackrel{def}{=} \{\ \ell \in I_s \mid p \stackrel{\ell}{\Rightarrow}\ \} \\ \boldsymbol{in}(P) &\stackrel{def}{=} \bigcap \{\ \boldsymbol{in}(q) \mid q \in P\ \} \end{aligned}$$

Definition 10. *Let $\sigma \in L^{\delta *}$; $\mathcal{L} \subseteq L^{\delta}$; and $\ell \in L^{\delta}$. Projecting a trace to a smaller set of labels is defined as:*

$$\begin{aligned} \epsilon \upharpoonright \mathcal{L} &\stackrel{def}{=} \epsilon \\ (\sigma \cdot \ell) \upharpoonright \mathcal{L} &\stackrel{def}{=} \begin{cases} (\sigma \upharpoonright \mathcal{L}) \cdot \ell & \text{if } \ell \in \mathcal{L} \\ \sigma \upharpoonright \mathcal{L} & \text{otherwise} \end{cases} \end{aligned}$$

Definition 11. *Let $s, e \in \mathcal{LTS}$ be **composable**, then s **accepts** e iff:*

$$\begin{aligned} s \leftharpoonup e \quad \stackrel{def}{=} \quad & \forall \sigma \in \boldsymbol{Utraces}(s \,||\, e),\ s' \in Q_s,\ e' \in Q_e : \\ & s \,||\, e \stackrel{\sigma}{\Rightarrow} s' \,||\, e' \implies \boldsymbol{out}(e') \cap I_s \subseteq \boldsymbol{in}(s') \cap U_e \end{aligned}$$

The symmetric version of the $\leftharpoonup$ relation is defined in Definitions 12. Though it might look like an equivalence relation, it is neither reflexive nor transitive. Reflexivity fails because $\leftrightharpoons$ is indirectly defined using parallel composition. This means it is only defined on specifications that are composable, and any specification with outputs is not composable with itself. Transitivity is also not true, because each pair of specifications has its own sets of state pairs and shared labels for which the $\leftharpoonup$ relation must hold. This means each specification pair must be checked independently of any other specifications.

Definition 12. *Let $s, e \in \mathcal{LTS}$ be* ***composable****, then s* ***mutually accepts*** *e:*

$$s \leftrightharpoons e \quad \stackrel{def}{=} \quad s \leftharpoonup e \ \wedge \ e \leftharpoonup s$$

5 Compositionalility for Uioco

The previous section defined what it means for a specification to not trigger undefined behaviour in another specification using the $\leftharpoonup$ relation. This section will prove that this property allows compositional testing using **uioco**.

Lemma 2 shows how for composable, input complete systems, traces in the composed system can be transformed into traces in the component systems, and the other way around. This allows for compositional model-based testing in input complete systems. Lemma 3 then goes on to show that for **Utraces**, the same is also possible as long as the two specifications are mutually accepting.

This is also where the **composable** requirement becomes important. It enforces that all labels are either synchronised or only present in one of the two label sets. This means that every trace σ can be split into a unique pair of two projected traces $\sigma \upharpoonright L_s^\delta$ and $\sigma \upharpoonright L_e^\delta$ which can be replayed in s and e, respectively. Without this requirement, it would be unclear what to do with unsynchronised shared labels.

Lemma 2. *let i_s, i_e be* ***composable*** *IOTS, $i'_s \in Q_{i_s}$, $i'_e \in Q_{i_e}, \sigma \in {L^\delta_{i_s \| i_e}}^*$.*

$$i_s \,||\, i_e \stackrel{\sigma}{\Rightarrow} i'_s \,||\, i'_e \iff i_s \xRightarrow{\sigma \upharpoonright L^\delta_{i_s}} i'_s \ \wedge \ i_e \xRightarrow{\sigma \upharpoonright L^\delta_{i_e}} i'_e$$

Lemma 3. *let s, e be* ***composable*** *LTS, $s' \in Q_s$, $e' \in Q_e, \sigma \in$ **Utraces**$(s \,||\, e)$.*

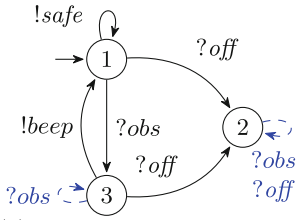
$$s \leftrightharpoons e \implies \left(s \,||\, e \stackrel{\sigma}{\Rightarrow} s' \,||\, e' \iff s \xRightarrow{\sigma \upharpoonright L^\delta_s} s' \ \wedge \ e \xRightarrow{\sigma \upharpoonright L^\delta_e} e' \right)$$

The $\leftrightharpoons$ relation, and by extension Lemma 3, only consider **Utraces** and not arbitrary traces because only states reachable by **Utraces** are important for **uioco**. This does, however, create the extra requirement of checking that after projecting a trace to a specific component, it is still part of the **Utraces** for that component. Lemma 4 shows that the $\leftrightharpoons$ relation ensures that **Utraces** are preserved when projecting from a composed system, in both directions. This is not trivial, as the special label δ is not normally preserved under composition.

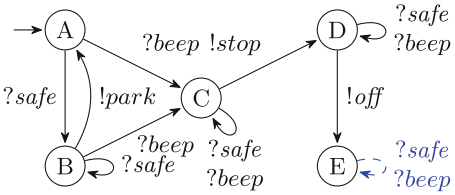
Lemma 4. *Let $s, e \in LTS$ be* ***composable****, $\sigma \in {L^\delta_{s||e}}^*$.*

$$s \leftrightharpoons e \implies$$
$$\left(\sigma \in \textbf{\textit{Utraces}}(s \,||\, e) \iff \sigma \upharpoonright L^\delta_s \in \textbf{\textit{Utraces}}(s) \ \wedge \ \sigma \upharpoonright L^\delta_e \in \textbf{\textit{Utraces}}(e) \right)$$

We now present Theorem 1, which is the main statement of this paper. It states that for mutually accepting specifications, **uioco** is preserved under parallel composition. The other way around, if there is a problem that causes two



(a) Adapted S_3 and I_3 : car sensor component



(b) Adapted S_4 and I_4: automated parking component

Fig. 5. Mutually accepting versions of Fig. 3. (spec: →, imp: -→)

composed implementations to be **uioco**-incorrect to their composed specifications, then this problem can also be found by testing with at least one of the components. The reverse of this implication, however, does not hold, even if both specifications are mutually accepting.

The reason for this is that the mutual acceptance relation only guarantees that no invalid outputs are communicated. It does not enforce that something is actually communicated. Therefore, it is possible for one of the two implementations to produce quiescence when this is not allowed, which is then masked in the combined system by the outputs generated by the other component. This highlights a property of the **uioco** relation: presence of specific outputs cannot be enforced. One possible way to deal with this might be to extend the **uioco** theory with a more fine grained concept of quiescence, allowing the detection of quiescence in specific components, instead of only over the whole system. This is further explored in [17].

Theorem 1. *Let* $s, e \in LTS$ *be* ***composable****,* $i_s, i_e \in IOTS$*, then*

$$s \leftrightharpoons e \ \wedge \ i_s \ \textbf{\textit{uioco}} \ s \ \wedge \ i_e \ \textbf{\textit{uioco}} \ e \implies i_s \,||\, i_e \ \textbf{\textit{uioco}} \ s \,||\, e$$

Another thing to note is that when applying Theorem 1 in practice, this makes the implicit assumption that you can correctly compose components. In order to guarantee the correctness of the composed system, the composition of components i_s and i_e must actually behave as $i_s \,||\, i_e$. This means that any communicating channels must be connected as described in s and e, and that there must not be some hidden implicit environment part of the composition setup that further influences the behaviour of either of the components.

6 The Parking System Revisited

Using the results from Sect. 4 and Sect. 5, the problems with the parking system from Sect. 3 can be explained: the two specifications in Fig. 3 are not mutually accepting. A counterexample is the trace $safe \cdot obs$, which is in the **Utraces** of $S_1||S_2$, and goes to state $B3$. In state 3, however, S_1 can perform output *beep*,

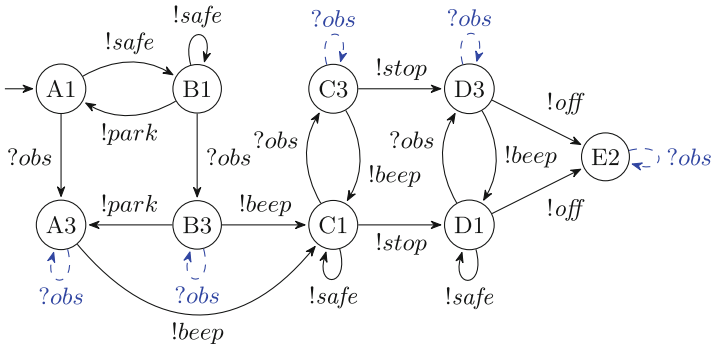


Fig. 6. Adapted car autopark and sensor composed ($S_3||S_4$ →) and ($I_3||I_4$ -↛)

while S_2 does not accept input *beep* in state B. A number of other counterexamples can also be given, each one corresponding to one of the dashed output transitions of $S_1||S_2$. These are the states where the composed implementation produces unspecified outputs. Using these counterexamples, the points where the specifications have to be extended can be identified. The result can be seen in Fig. 5. Figure 5b now has several extra transitions for *safe* and *beep* defined in the specification, exactly in those places where the sensor might supply these inputs. The developer is now forced to think about what actually should happen there, while the developer is still free to not specify inputs that should not occur in normal operation. This is especially relevant for the *beep* transition originating from state B, which was previously unspecified. On further inspection, it is revealed that a simple self loop is not desired here, because after a *beep* the car should stop, and not continue to park. This would have resulted in undesired behaviour if the specifications were simply made input enabled in an automatic way, as was done in previous approaches [4]. Using a self-loop here would mean that implementations are possible which pass all tests, but still do not stop when an object is detected.

The result of composing the adapted specifications from Fig. 5 is shown in Fig. 6. This specification now correctly finds that $I_1||I_2$ ~~**uioco**~~ $S_3||S_4$, which can be seen with the trace $safe \cdot obs \cdot beep$ which is present in the **Utraces** of $S_3||S_4$. After this trace, $I_1||I_2$ can produce the output *park*, which is undesirable after detecting an object, and also not allowed by $S_3||S_4$. But if each individual implementation is updated to be **uioco** correct according to its own adapted specification, as is done with I_3 and I_4, then their composition is again correct with respect to the composed specifications, i.e. $I_3||I_4$ **uioco** $S_3||S_4$.

The example shows how the $\leftrightharpoons$ relation can be used to find integration problems between components using their specifications. Possible problems are prevented by expanding the specification, without requiring a full specification of all inputs. This does not yet require any actual implementations, as the reasoning is done over the domain of all possible valid implementations. Finding these

integration problems before starting integration testing, allows for fixing them earlier in development.

7 Component Substitution and Diagnosis

In addition to providing compositionality in development and testing, mutual acceptance has benefits when retesting systems with updated components, and when diagnosing systems consisting of components. A common situation is that one component becomes deprecated and needs to be replaced. This traditionally has a high cost, because even if the new component is well tested, there is a chance using it will cause problems with the other components already in use. These issues mainly occur because replacing a component changes the environment for the other components. This means the other components, which are the environment of the replaced component, might be called with new inputs which have not yet been tested. A well known example where reuse of an old, well tested component in a new environment caused the whole system to fail is the crash of the Ariane-5 rocket [15,21]. Here, an important subsystem put implicit requirements on the environment which were not documented or checked to hold. Correctness was inferred from extensive testing, but after changing the environment this testing became invalid, and the component failed anyway.

These problems can be reduced by using a specification-based analysis like $\leftrightharpoons$ in combination with model-based testing. Model-based testing can generate tests for every defined sequence of inputs. If two specifications are mutually accepting, then they only communicate outputs which are defined inputs for the intended communication partner. These two points together mean that all the model-based testing done up to the point of replacing a component is still useful, because it was testing for all possible inputs, and not just the ones that were in current use. This can give a much higher confidence that a component switch will not cause any problems, because testing does not have to start from square one. If the specification of the new component is not mutually accepting with all the rest of the system, then the counterexamples point to all the places where undefined inputs are given. This information can be used to improve the specifications, and focus testing toward these possible problem areas.

The correctness reasoning made possible by the $\leftrightharpoons$ relation can also be used during diagnosis, by taking the converse of Theorem 1. If the whole system contains a problem, and one or more components are found to be **uioco** correct, then the problem must be located within one of the remaining components. Together with Lemmas 3 and 4 this can then be used to narrow down a trace showing **uioco** incorrectness of the whole system to a shorter trace showing **uioco** incorrectness of one specific component. This idea is expressed in Lemma 5. Since **composable** requires each label to be part of at most one output set, the last output of the counterexample uniquely identifies the problem component. This does not work if the last output was δ, which could have been caused by a number of components. In this case we can still find the faulty component by replaying the projected traces in all components until the faulty one is found. This can,

for example, more accurately determine the source of bugs from gathered logs containing full system traces.

Lemma 5. *Let* $s, e \in LTS$, $i_s, i_e \in \mathcal{IOTS}$, $s \leftrightharpoons e$, $\sigma \in \textbf{\textit{Utraces}}(s \,||\, e)$.

$$\begin{array}{l} \sigma \textit{ is a counterexample for } i_s \,||\, i_e \textbf{ \textit{uioco}} \; s \,||\, e \implies \\ \quad \sigma \upharpoonright L_s^\delta \textit{ is a counterexample for } i_s \textbf{ \textit{uioco}} \; s \quad \vee \\ \quad \sigma \upharpoonright L_e^\delta \textit{ is a counterexample for } i_e \textbf{ \textit{uioco}} \; e \end{array}$$

8 Related Work

The work in this paper is closely related to ideas already discussed in the context of interface automata [8–10]. Interface automata are a type of labelled transition system which can be used to model both the behaviour of a component and the constraints it puts on its environment. These constraints are encoded in the form of missing input transitions, which then signify that the component can only be used in an environment that does not give these inputs. This closely resembles the main idea behind the $\leftrightharpoons$ relation. Apart from a slightly different composability requirement which makes it associative, our definition for parallel composition coincides with the one from [8]. Our definition for $\leftrightharpoons$ also seems to coincide with the absence of reachable (by **Utraces**) error states as defined in [8]. The solution to reachable error states taken for interface automata is to apply a pruning algorithm. This will remove input transitions to further restrict the valid environments until all error states become unreachable. A downside of this approach is that it becomes easy to generate composed models that after pruning no longer give errors, but also no longer express the desired correct behaviour. This is noted in [8] as the observation that the environment that does not give any inputs at all, always avoids all avoidable error states. The interface automata approach consists of removing transitions from the composed specification until problem areas are unreachable. We instead choose to add transitions to the component specifications until the problem areas no longer exist. Another contribution of our work is the inclusion of quiescence, and the direct link to the **uioco** implementation relation. This makes the theory easier to apply in practice in the context of existing MBT tools.

An earlier attempt at formalising the correctness of a component with respect to its environment was developed in [11]. It defines the **eco** (environmental conformance) relation with similar semantics to the accepts relation. The relation **eco**, however, works on a specification for the environment, and a black box implementation of the component. This means that **eco** conformance can only be checked by testing, and this needs to be redone completely whenever a component changes. Additionally, all labels of the component and its environment have to communicate, i.e., there is no external communication, which further restricts applicability. The **eco** approach also has a couple of advantages. Since **eco** is checked using testing, it can be done on the fly. It also does not require how a component calls other components as part of its input specifications. Instead,

this information is gathered while testing and compared against the specifications of the components being called. This makes a possible combination of our work with the **eco** theory and algorithms interesting.

In this paper, we describe when a component is a valid environment for another component. Earlier work looking into the set of valid environments for a given component was done in the field of contract-based design. A detailed overview of this field can be found in [2]. A contract is defined as a tuple of a set of valid environments and a set of valid implementations, where every combination of environment and implementation can be composed. The definition of what it means for an environment to be composable with an implementation is very similar to our definitions, and it also describes how a labelled transition system can be seen as a contract. The scope under consideration in [2], however, is limited to receptive environments with the same label set as the components. All components also have to be deterministic, and internal transitions or quiescence are not discussed. A more recent addition to contract theory extends the scope of [2] to hyper-contracts [13]. While this extends the scope of properties that can be expressed as contracts, the current instantiation of the meta-theory for labelled transition systems still has many of the restrictions imposed in [2]. In contrast to the bottom up approach of combining component contracts into a composed contract, a top down approach is also possible and sometimes desired. Decomposing a set of requirements into individual component contracts has been studied in [14].

Another way of describing compatible components is defining a specification for the most permissive communication partner. All concrete communication partners are then in some form of a refinement relation with this "operating guideline". This approach is outlined in [16] for acyclic finite labelled transition systems. It assumes all communications to be asynchronous, while we assume synchronous communication.

9 Future Work

Making specifications mutually accepting involves defining extra behaviour. Some of this extra specification is desirable, for instance the *beep* transition from state B in S_4. This transition represents interesting behaviour that was missed in the specification phase. Most other added transitions, however, are just simple self-loops, which represent that the input has to be ignored. If receiving an input that was not specified is considered undefined behaviour, this is required to ensure correct behaviour. Another possible interpretation would be that unspecified inputs are buffered, until the other component is ready to receive them. In such a setting, it would not be required that every input that can be given is specified immediately. It would then be enough that such inputs are specified always eventually, after some amount of internal actions of the receiving component. In general, it can be investigated how to (automatically) repair non-mutually accepting systems.

In this paper, we have defined mutual acceptance, but no ways for practically checking it have been given. Algorithms to efficiently check mutual acceptance

between specifications, or testing procedures to test mutual acceptance, analogous to **eco**, need to be developed.

The theory introduced so far works on two components. Larger systems consist of many components. Mutual acceptance can still be inferred by repeatedly applying the parallel composition operator and Theorem 1. For example, when combining specifications $s1, s2$ and $s3$ into $(s1 \mid\mid s2) \mid\mid s3$, we must check that $s1 \mid\mid s2 \leftrightharpoons s3$. Doing this directly using $s1 \mid\mid s2$ might be complicated due to the increasing number of parallel components. We postulate that multiway-mutual acceptance can be inferred from pairwise-mutual acceptance. In general, mutual acceptance of many components, with complicated communication structures, should be further investigated.

Requiring $\leftrightharpoons$ for all intermediate steps means that there cannot be any unexpected outputs. For real systems however, these outputs are only a problem if they appear in the final composition of all the components. The fact that two components do not work well in all environments is not a problem if you plan to use them together with other components that will prevent this. Therefore, a different definition of mutual acceptance for more than two components at a time might be investigated.

To apply the theory in this paper to a practical use case, it will need to be extended with the concept of data. Real systems can seldom be modelled with a finite set of labels, but will instead send instances of data types to each other. This has been formalised in the theory of symbolic transition systems (STS) [5,12], which is the underlying formalism of several MBT tools. The concepts in this paper could be extended to STS which would bring them closer to being applied in practice.

10 Conclusion

Model-based testing is a promising technology for increasing the efficiency and effectiveness of testing. The applicability of MBT, however, is limited by the availability of models. Larger system models are hard to create, but can be composed from multiple smaller component models. In this paper, we have defined the mutual acceptance relation $\leftrightharpoons$ between specifications, which guarantees that model-based testing is compositional, i.e. if two components have been tested for **uioco**-correctness with respect to their respective specifications, then the composition of these implementations is also **uioco**-correct with respect to the composition of their specifications, under the assumption that the parallel composition operator itself is faithfully implemented. This is an improvement over previous results which obtained the same conclusion with a stricter requirement, viz. that all specifications must be input-enabled [4]. In addition, we have shown that this result can also help when updating older components with newer ones, and when localising a faulty component during diagnosis of a large, component-based system.

References

1. Beneš, N., et al.: Complete composition operators for IOCO-testing theory. In: Proceedings of the 18th International ACM SIGSOFT Symposium on Component-Based Software Engineering. CBSE 2015. New York, NY, USA, May 2015, pp. 101–110. Association for Computing Machinery (2015). ISBN: 978-1-4503-3471-6. https://doi.org/10.1145/2737166.2737175. Accessed 11 Oct 2021
2. Benveniste, A., et al.: Contracts for system design. Found. Trends ® Electron. Des. Autom. **12**(2-3), 124–400 (2018). ISSN: 1551-3939, 1551-3947. https://doi.org/10.1561/1000000053. http://www.nowpublishers.com/article/Details/EDA-053. Accessed 20 Dec 2022
3. Berendsen, J., Vaandrager, F.: Composition in a Paper by de Alfaro e.a. Is Not Associative. Technical report. Radboud University, May 2008 (2008). https://sws.cs.ru.nl/publications/papers/fvaan/commentdAdSF+.pdf
4. van der Bijl, M., Rensink, A., Tretmans, J.: Compositional testing with IOCO. In: Petrenko, A., Ulrich, A. (eds.) FATES 2003. LNCS, vol. 2931, pp. 86–100. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-24617-6_7 ISBN: 978-3-540-24617-6. Accessed 29 Oct 2021
5. van den Bos, P., Tretmans, J.: Coverage-based testing with symbolic transition systems. In: Beyer, D., Keller, C. (eds.) TAP 2019. LNCS, vol. 11823, pp. 64–82. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-31157-5_5 ISBN: 978-3-030-31156-8. Accessed 23 Fen 2023
6. van Cuyck, G., van Arragon, L., Tretmans, J.: Compositionality in model-based testing. In: CoRR abs/2307.03701 (2023). arXiv: 2307.03701. https://doi.org/10.48550/arXiv.2307.03701
7. Daca, P., et al.: Compositional specifications for IOCO testing. In: Proceedings - IEEE 7th International Conference on Software Testing, Verification and Validation, ICST 2014. March 2014, pp. 373–382 (2014). https://doi.org/10.1109/ICST.2014.50. ISBN: 978-1-4799-2255-0
8. de Alfaro, L., Henzinger, T.A.: Interface automata. In: ACM SIGSOFT Software Engineering Notes 26(5), 109–120 (2001). ISSN: 0163–5948. https://dl.acm.org/doi/10.1145/503271.503226. https://doi.org/10.1145/503271.503226. Accessed 29 Oct 2021
9. de Alfaro, L., Henzinger, T.A.: Interface theories for component-based design. In: Henzinger, T.A., Kirsch, C.M. (eds.) EMSOFT 2001. LNCS, vol. 2211, pp. 148–165. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-45449-7_11 ISBN: 978-3- 540-42673-8. Accessed 29 Oct 2021
10. de Alfaro, L., Henzinger, T.A.: Interface-based design. In: Broy, M., Grünbauer, J., Harel, D., Hoare, T. (eds.) Engineering Theories of Software Intensive Systems. NSS, vol. 195, pp. 83–104. Springer, Dordrecht (2005). https://doi.org/10.1007/1-4020-3532-2_3 ISBN: 978-1-4020- 3532-6
11. Frantzen, L., Tretmans, J.: Model-based testing of environmental conformance of components. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2006. LNCS, vol. 4709, pp. 1–25. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-74792-5_1 ISBN: 978-3-540-74792-5
12. Frantzen, L., Tretmans, J., Willemse, T.A.C.: Test generation based on symbolic specifications. In: Grabowski, J., Nielsen, B. (eds.) FATES 2004. LNCS, vol. 3395, pp. 1–15. Springer, Heidelberg (2005). https://doi.org/10.1007/978-3-540-31848-4_1 ISBN: 978-3-540-25109-5. Accessed 3 Feb 2023

13. Incer, I., et al.: From interface automata to hypercontracts. In: Raskin, J.F., Chatterjee, K., Doyen, L., Majumdar, R. (eds.) Principles of Systems Design. LNCS, vol. 13660, pp. 477–493. Springer, Cham (2022). https://doi.org/10.1007/978-3-031-22337-2_23. ISBN: 978-3-031-22337-2. Accessed 10 Jan 2023
14. Kaiser, B., et al.: Contract-based design of embedded systems integrating nominal behavior and safety. In: Complex Systems Informatics and Modeling, October, pp. 66–91 (2015). https://doi.org/10.7250/csimq.2015-4.05
15. Lions, J.L.: Ariane 5: Flight 501 Failure. Technical Report (1996). https://esamultimedia.esa.int/docs/esa-x-1819eng.pdf
16. Massuthe, P., Schmidt, K.: Operating guidelines - an automata-theoretic foundation for the service-oriented architecture. In: Fifth International Conference on Quality Software (QSIC 2005). September 2005, pp. 452–457. https://doi.org/10.1109/QSIC.2005.47
17. Noroozi, N.: Improving Input-Output Conformance Testing Theories. Ph.D. thesis. Eindhoven: Technische Universiteit Eindhoven, October (2014). https://wiki.hh.se/ceres/images/c/c2/Noroozi_thesis_2014.pdf. Accessed 15 Feb 2023
18. Tretmans, J.: Model Based Testing with Labelled Transition Systems. In: Hierons, R.M., Bowen, J.P., Harman, M. (eds.) Formal Methods and Testing. LNCS, vol. 4949, pp. 1–38. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78917-8_1 ISBN: 978-3-540-78917-8. Accessed 22 Dec 2022
19. Tretmans, J., Janssen, R.: Goodbye IOCO. In: Jansen, N., Stoelinga, M., van den Bos, P. (eds.) A Journey from Process Algebra via Timed Automata to Model Learning. LNCS, vol. 13560, pp. 491–511 (2022). https://doi.org/10.1007/978-3-031-15629-8_26. ISBN: 978-3-031-15628-1. Accessed 4 Oct 2022
20. Vogler, W., Lüttgen.: A Linear-Time Branching-Time Per-spective on Interface Automata. Acta Informatica **57**(3), pp. 513–550 (2020). https://doi.org/10.1007/s00236-020-00369-4. ISSN: 1432-0525, Accessed 21 Jun 2022
21. Weyuker, E.J.: Testing component-based software: a cautionary tale. IEEE Software **15**(5), 54–59. https://doi.org/10.1109/52.714817. ISSN: 1937–4194