Observer-based Decomposition Assuming Known Interfaces*

Bharat Garhewal[1]

Technische Universiteit Eindhoven, The Netherlands
b.garhewal@student.tue.nl

Abstract. Decomposition of monolithic models is useful in re-factoring and re-engineering of software systems. Externally, a software system behaves as a single (monolithic) unit, while internal behaviour is specified by multiple components working together, where each component may or may not know the status of the other components. We propose an algorithm to decompose a monolithic specification model represented as a Labelled Transition System (LTS) into a set of known interfaces (also LTSs) and a controller, which consists of an observer and a set of state-based event constraints. A model bisimilar to the specification is generated when the controller is combined with the known interfaces.

Our approach assumes that state information of the interfaces is visible to the controller, restricting the events not allowed by the specification by means of state-based event constraints. If state-information is insufficient to adequately control the behaviour of the interfaces, an observer is generated. In some cases, different choices of observers are possible, because the same information regarding the state of the system can be obtained by tracking different events. We illustrate this by an example, and argue that different observers lead to different interpretations of the overall system. A method to steer which decomposition is chosen, however, remains as future work.

Keywords: decomposition · labelled transition system · observer · state-based event constraints

1 Introduction

Decomposition of monolithic software models is highly desirable for reasons like reducing complexity, ease of verification, etc. Monolithic models can be too complex to comprehend, and consequently, implement or modify. Therefore, decomposition of software models is a well-studied topic [1, 2, 4–7, 9, 13–15]. Clearly, a decomposition is useful only when it fulfils its requirements: for example, the Krohn-Rhodes Theorem [9] – at risk of over-simplification – could be used to provide a 'hierarchical' decomposition of a deterministic automaton. Work on

* Thanks are due to my assessment committee and my friends who listened/helped me during my work. There are too many to name, but you know who you are!

decomposition of specifications written as LOTOS (Language Of Temporal Ordering Specification) processes [2, 6, 8] is focused towards generating LOTOS sub-processes from a single process and a bi-partition of events, where each partition of events captures a group of 'related' functionality. In another case, decomposition in Event-B [4, 15] is used in top-down refinement, where an abstract model can be successively refined into multiple sub-models given a partition of events or states (encoded as variables in Event-B).

One source of monolithic models is Automata Learning (AL) [16] - the process of learning software models as automata. AL typically treats a system as a black-box, purely observing the inputs and outputs of the system and learning its behaviour as an automaton. Typical AL algorithms produce a monolithic model, as they are not concerned with the internal composition of the system under learning (some exceptions are [10, 11] which deal with modularity in systems).

Consider a system consisting of the set of components $\{I_1, I_2, \cdots, I_N\}$. The combined behaviour of the components is defined by (system) specification S. There exists a 'hidden' control component (aka *controller*) $\mathcal{C}$, responsible for coordinating the other components. Controller $\mathcal{C}$ can 'peek' into the state-space of the other components; thus, the component models are called *interfaces* from the perspective of the controller $\mathcal{C}$. $\mathcal{C}$ imposes restrictions on the behaviour of the interfaces using state-based event constraints $\mathcal{R}$. If state-information is insufficient to adequately control the components (i.e., the specification $\mathcal{S}$ distinguishes states which the controller $\mathcal{C}$ cannot), we add an observer $\mathcal{O}$ to provide the controller $\mathcal{C}$ with supplementary state-information from specification $\mathcal{S}$. The described architecture is visualised in Figure 1.

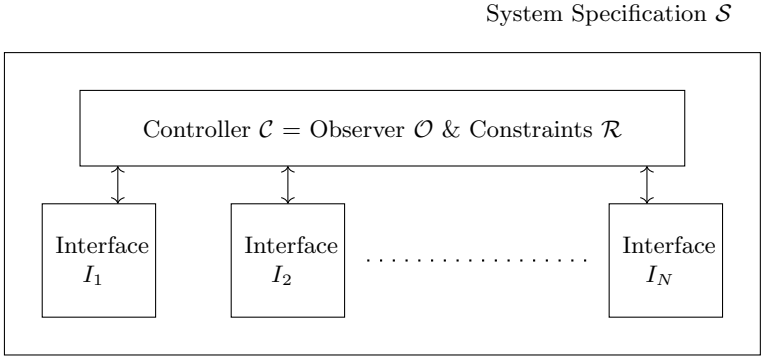


Fig. 1: Component-based view of a system: Specification $\mathcal{S}$ is a result of a 'hidden' control component $\mathcal{C}$ coordinating behaviour of the set of interfaces $\{I_1, I_2, \cdots, I_N\}$. An interface represents a 'view' of a component visible to the control component $\mathcal{C}$. Our goal is to find a satisfying control component $\mathcal{C}$, where $\mathcal{C}$ consists of an observer $\mathcal{O}$ to record historical information and state-based event constraints $\mathcal{R}$ to restrict behaviour of the interfaces.

An observer is a construct used to record historical behaviour (i.e., traces used to reach some state). As there may be multiple ways to do so, each possibly having its own interpretation, our approach allows the generation of non-unique observers where we try to generate a 'minimal' observer (i.e., an observer which records as little as possible, specifically, having fewer states). Consequently, our approach is focused on generating a 'locally minimal' observer. Each locally minimal observer may provide differing insights into the decomposition (all equally valid), with 'more insightful' being dependent on the domain and/or individual using the observer. Steering the decomposition process further would require insight into what is a 'good' observer, and is not considered in our work.

To the best of our knowledge, existing decomposition techniques do not solve our problem completely, as our case has known interface models and allows common (shared) events. Therefore, we need an algorithm that - given a monolithic model and its interface models - automatically generates the aforementioned controller $\mathcal{C}$ to co-ordinate behaviour between the interface models. In such a case, the behaviour of the controller and interfaces should be bisimilar to the original specification.

Our decomposition approach may be considered as a problem from the perspective of Supervisory Control Theory (SCT) [12]. SCT – while not a decomposition technique – is applicable in the context of our work, as it can generate a controller (called 'supervisor' in SCT) given a (set of) plant automaton and a (set of) requirements. A direct application of supervisory control theory, with the set of interfaces as a plant and the target system as a specification is not an ideal solution because, while it gives us a valid solution for a controller, the generated observer is (almost) identical to the original specification. If an observer is identical to the original specification, we may duplicate existing state-information from the interface models. Section 4 compares our work to SCT in detail.

In this paper, we present two algorithms: the first is a depth-first algorithm to generate a locally minimal observer[1] for a monolithic model, and the second to generate state-based event constraints to restrict the behaviour of the interfaces of the monolithic model. We formally prove the correctness of our approach and apply the algorithms to an academic example in order to demonstrate the capabilities of our approach. As the algorithms are currently implemented as a poof-of-principle, their application for larger monolithic models is yet to be investigated. However, we apply our algorithms on an example to demonstrate that different observers lead to different interpretations of the original system.

The remainder of this paper is structured as follows: Section 2 contains preliminary definitions; Section 3 formalises the problem statement using terminology defined in the preliminaries; Section 4 discusses our work in the context of relevant literature and supervisory control; Section 5 discusses our methodology and associated proofs; Section 6 applies our approach to an example showing decomposition with two different observers; finally, we conclude in Section 7.

[1] The current implementation can generate a subset of the theoretically possible solution-space, i.e., a subset of all possible satisfying observers.

2 Preliminary definitions and constructions

Definition 1 (Labelled Transition System). *A Labelled Transition System is a tuple* $(Q, \Sigma, \rightarrow, q_0)$ *where* Q *is a finite set of states with* $q_0 \in Q$ *as initial state;* Σ *is a finite set of events, called the* alphabet *of the system; and* $\rightarrow$ *is a (partial) transition relation* $\rightarrow \subseteq Q \times \Sigma \times Q$.

If the transition relation $\rightarrow$ is functional, the Labelled Transition System (LTS) is called a Deterministic Labelled Transition System (DLTS), otherwise it is called a Non-deterministic Labelled Transition System (NLTS). We say that an event e at state s is defined if there exists a state s' such that $(s, e, s') \in \rightarrow$; we denote this as $s \xrightarrow{e}$; if there does not exist such an s', we write $s \not\xrightarrow{e}$.

An LTS $\mathcal{A}$ generates a word $\sigma \in \Sigma_{\mathcal{A}}^*$, with $\sigma = e_1 e_2 \dots e_n$, if there exist $s_0, s_1, \dots, s_n \in Q_{\mathcal{A}}$ such that $(s_i, e_{i+1}, s_{i+1}) \in \rightarrow_{\mathcal{A}}$ for all $0 \leq i < n$. The set of all such words is the language of $\mathcal{A}$, denoted as $\mathcal{L}(\mathcal{A})$. We omit the subscript $\mathcal{A}$ from $\rightarrow_{\mathcal{A}}$ when it is clear from context.

We define a *flower* labelled transition system as a special case of LTS.

Definition 2 (Flower Labelled Transition System). *A flower (or unit) transition system is a finite LTS* $(Q, \Sigma, \rightarrow, q_0)$ *where* $Q = \{q_0\}$, *and* $\rightarrow = Q \times \Sigma \times Q$.

A flower LTS is denoted using 1_Σ where Σ is the alphabet of the LTS; however, we sometimes write 1_X where X is an LTS to indicate a flower LTS with the alphabet of X. The language of a flower LTS $\mathcal{A}$ is equal to the closure of the alphabet of $\mathcal{A}$, i.e., $\mathcal{L}(\mathcal{A}) = \Sigma_{\mathcal{A}}^*$.

Next, we define a constrained parallel composition operator.

Definition 3 (Constrained Parallel Composition). *The parallel composition of LTSs* $A = (Q_A, \Sigma_A, \rightarrow_A, q_{0A})$ *and* $B = (Q_B, \Sigma_B, \rightarrow_B, q_{0B})$ *constrained by* $\mathcal{R} \subseteq (\Sigma_A \cup \Sigma_B) \times Q_A \times Q_B$ *is defined as:*

$$A \parallel_{\mathcal{R}} B = (Q_A \times Q_B, \Sigma_A \cup \Sigma_B, \rightarrow, (q_{0A}, q_{0B})),$$

where $\rightarrow \subseteq (Q_A \times Q_B) \times (\Sigma_A \cup \Sigma_B) \times (Q_A \times Q_B)$ *is the transition relation. The transition* $(q_1, q_2) \xrightarrow{e} (q_1', q_2')$ *is contained in* $\rightarrow$ *iff the transition satisfies one of the following properties:*

1. $q_1 \xrightarrow{e}_A q_1'$, $q_2 \xrightarrow{e}_B q_2'$, $e \in \Sigma_A \cap \Sigma_B$, *and* $(e, (q_1, q_2)) \notin \mathcal{R}$, *or*
2. $q_1 \xrightarrow{e}_A q_1'$, $q_2 = q_2'$, $e \in \Sigma_A \setminus \Sigma_B$, *and* $(e, (q_1, q_2)) \notin \mathcal{R}$, *or*
3. $q_1 = q_1'$, $q_2 \xrightarrow{e}_B q_2'$, $e \in \Sigma_B \setminus \Sigma_A$, *and* $(e, (q_1, q_2)) \notin \mathcal{R}$.

Intuitively, the definition constrains events according to the 'current' state of the LTSs A and B. If an event e is defined at state $(q_a, q_b) \in Q_A \times Q_B$ of an unconstrained composition and $(e, (q_a, q_b)) \in \mathcal{R}$, then e is blocked at state (q_a, q_b) and cannot occur. It is recommended to consider blocking of events by constraints as a form of 'post-processing' the result of the unconstrained parallel composition operator, rather than considering blocking from a compositional

perspective. While the latter is correct, the former makes it easier to follow the intuition of our work.

If $\mathcal{R} = \varnothing$, the unconstrained parallel composition operator becomes equivalent to the "synchronous composition" defined in [3]. The unconstrained parallel composition operator is used to describe the possible behaviour of two LTSs working concurrently. The operator synchronises on shared events, i.e., both LTSs must perform the shared event simultaneously (line 1 of the transition relation). Events which are not shared between the two LTSs are allowed to occur asynchronously (lines 2 and 3 of the transition relation).

Next, we define a simulation relation between two LTSs.

Definition 4 (Simulation Relation). *A relation $R \subseteq Q_S \times Q_T$ between the states of LTSs $S = (Q_S, \Sigma_S, \rightarrow_S, q_s)$ and $T = (Q_T, \Sigma_T, \rightarrow_T, q_t)$, is a* simulation relation *from S to T if $\Sigma_T \subseteq \Sigma_S$ and*

1. *$q_s R q_t$;*
2. *for all $s, s' \in Q_S$, $t \in Q_T$ and $e \in \Sigma_S \cap \Sigma_T$ with $s \xrightarrow{e}_S s'$ and sRt, there exists a $t' \in Q_T$ such that $t \xrightarrow{e}_T t'$ and $s'Rt'$;*
3. *for all $s, s' \in Q_S$, $t \in Q_T$ and $e \in \Sigma_S \backslash \Sigma_T$ with $s \xrightarrow{e}_S s'$ and sRt, we find $s'Rt$.*

A simulation relation R from S to T is *total* if for every $s \in Q_S$ there is a $t \in Q_T$ such that sRt. A simulation relation is *injective* if for any two $s, s' \in Q_S$ and $t \in Q_T$ with sRt and $s'Rt$ we find $s = s'$. If all states in S are reachable from the initial state, any simulation relation relation to another T will be total. In this paper, we assume that all states in an LTS are reachable, and hence, all simulation relations are total. A simulation relation $R \subseteq Q_S \times Q_T$ is a *bisimulation* relation if $R^{-1} \subseteq Q_T \times Q_S$ is also a simulation relation. We indicate bisimulation as $S \leftrightarrows T$.

Definition 4 is essentially a generalised version of simulation relations used elsewhere. Our definition of generalised simulation relation coincides with other definitions if, for a simulation relation $R \subseteq Q_S \times Q_T$ for two LTSs S and T, the alphabet of T is a super-set of the alphabet of S, i.e., $\Sigma_T \supseteq \Sigma_S$.

Following that, we show that the simulation relation is congruent for unconstrained parallel composition.

Theorem 1 (Congruence of simulation relation). *If there exist simulation relations $R_i \subseteq Q_{\mathcal{S}} \times Q_{I_i}$ for all $i \in [1, N]$, then there exists a simulation relation $R \subseteq Q_{\mathcal{S}} \times Q_{\mathcal{I}}$ where $\mathcal{I} = I_1 \parallel_{\varnothing} I_2 \parallel_{\varnothing} \cdots \parallel_{\varnothing} I_N$.*

Proof. It suffices to prove the theorem for $N = 2$; other cases follow by induction. Given simulation relations $R_1 \subseteq Q_{\mathcal{S}} \times Q_{I_1}$ and $R_2 \subseteq Q_{\mathcal{S}} \times Q_{I_2}$, we show that $R = \{(s, (i_1, i_2)) \,|\, (s, i_1) \in R_1 \textit{and } (s, i_2) \in R_2\}$ is a simulation relation from $\mathcal{S}$ to $\mathcal{I} = I_1 \parallel_{\varnothing} I_2$. We decompose the proof into three parts (for each property of the simulation relation) –

Initial state. Let q_s be the initial state of $\mathcal{S}$. As R_1 and R_2 are simulation relations, we know that $(q_s, q_{i1}) \in R_1$ and $(q_s, q_{i2}) \in R_2$ where q_{i1} and q_{i2} are the initial states of I_1 and I_2. Observe, (q_{i1}, q_{i2}) is the initial state of $\mathcal{I} = I_1 \parallel_{\varnothing} I_2$. Therefore, by construction, we conclude $(q_s, (q_{i1}, q_{i2})) \in R$.

The remaining two cases for events are $e \in (\Sigma_{\mathcal{S}} \cap \Sigma_{\mathcal{I}})$ and $e \in (\Sigma_{\mathcal{S}} \setminus \Sigma_{\mathcal{I}})$, which are handled below.

Event $\mathbf{e \in \Sigma_{\mathcal{S}} \cap (\Sigma_{I_1} \cup \Sigma_{I_2})}$. We provide proof by case distinction on e, which requires three cases: the first, $e \in \Sigma_{\mathcal{S}} \cap (\Sigma_{I_1} \cap \Sigma_{I_2})$; the second, $e \in \Sigma_{\mathcal{S}} \cap (\Sigma_{I_1} \setminus \Sigma_{I_2})$; and the third, $e \in \Sigma_{\mathcal{S}} \cap (\Sigma_{I_2} \setminus \Sigma_{I_1})$. The proof is as follows:

1. $\mathbf{e \in \Sigma_{\mathcal{S}} \cap (\Sigma_{I_1} \cap \Sigma_{I_2})}$: Let $s \xrightarrow{e} s'$ and $(s, i) \in R$, where $i \in Q_{\mathcal{I}}$. Using the definition of parallel composition, we know $i = (i_1, i_2) \subseteq Q_{I_1} \times Q_{I_2}$. As $(s, i_1) \in R_1$, $(s, i_2) \in R_2$, and $e \in \Sigma_{I_1} \cap \Sigma_{I_2}$, there exist i_1', i_2' such that $i_1 \xrightarrow{e} i_1'$ and $i_2 \xrightarrow{e} i_2'$ with $(s', i_1') \in R_1$ and $(s', i_2') \in R_2$. By definition of the parallel composition, we note $i \xrightarrow{e} i'$ where $i' = (i_1', i_2')$. Therefore, by construction, $(s', i') \in R$.
2. $\mathbf{e \in \Sigma_{\mathcal{S}} \cap (\Sigma_{I_1} \setminus \Sigma_{I_2})}$: Let $s \xrightarrow{e} s'$ and $(s, i) \in R$, where $i \in Q_{\mathcal{I}}$. Using the definition of parallel composition, we know $i = (i_1, i_2) \subseteq Q_{I_1} \times Q_{I_2}$. As $(s, i_1) \in R_1$, $(s, i_2) \in R_2$, and $e \in \Sigma_{I_1} \setminus \Sigma_{I_2}$, there exists i_1' such that $i_1 \xrightarrow{e} i_1'$ with $(s', i_1') \in R_1$. By definition of the parallel composition, we note $i \xrightarrow{e} i'$ where $i' = (i_1', i_2)$. Therefore, by construction, $(s', i') \in R$.
3. $\mathbf{e \in \Sigma_{\mathcal{S}} \cap (\Sigma_{I_2} \setminus \Sigma_{I_1})}$: Analogous to the previous case.

Event $\mathbf{e \in \Sigma_{\mathcal{S}} \setminus (\Sigma_{I_1} \cup \Sigma_{I_2})}$. Let $s \xrightarrow{e} s'$ and $(s, i) \in R$, where $i \in Q_{\mathcal{I}}$. As $(s, i_1) \in R_1$, $(s, i_2) \in R_2$, and $e \notin (\Sigma_{I_1} \cup \Sigma_{I_2})$, we can write $i \not\xrightarrow{e}$. We know that $(s', i_1) \in R_1$, $(s', i_2) \in R_2$ (definition of simulation relation) and therefore derive $(s', i) \in R$.

□

The above theorem proves that simulation relations are congruent for unconstrained parallel composition, we do not make any claim for constrained parallel composition.

3 Decomposition based on Interfaces

This section contains the formal description of our decomposition problem. We discuss a component with interfaces and a specification, and provide formal definitions for the same. Following that, in order to show how constraints and observers work, we present two decomposition sketches.

Internally, we assume that the system consists of a unknown controller $\mathcal{C}$ and multiple components which are visible to the controller as the set of interfaces $\{I_1, I_2, \cdots, I_N\}$. Controller $\mathcal{C}$ is a pair $(\mathcal{R}, \mathcal{O})$ where $\mathcal{R}$ is a set of constraints imposed on the behavior of the interfaces and $\mathcal{O}$ is an observer LTS. The decomposition problem is to find a pair $\mathcal{C} = (\mathcal{R}, \mathcal{O})$ such that the overall behaviour is equivalent to $\mathcal{S}$.

We now formally define an interface, followed by formalizing a specification.

Definition 5 (Interface). *An LTS I is an interface of an LTS $\mathcal{S}$ if and only if there is a simulation relation $R \subseteq Q_{\mathcal{S}} \times Q_I$. An interface is* complete *when the alphabet of I is equal to the alphabet of $\mathcal{S}$, i.e., $\Sigma_I = \Sigma_{\mathcal{S}}$.*

Following the definition of an interface, we can apply **??** to obtain the physical significance of the relationship between a specification and an interface: the specification may only perform behaviour which is allowed by its interfaces. If there exist simulation relations $R_i \subseteq Q_{\mathcal{S}} \times Q_{I_i}$ (for all $i \in [1, N]$), then there exists a simulation relation $R \subseteq Q_{\mathcal{S}} \times Q_{\mathcal{I}}$ where $\mathcal{I} = I_1 \parallel_{\varnothing} I_2 \parallel_{\varnothing} \cdots \parallel_{\varnothing} I_N$ (see Theorem 1). A complete interface simplifies the scope of our problem from multiple interfaces to a single interface. If there does not exist a simulation relation $R \subseteq Q_{\mathcal{S}} \times Q_{\mathcal{I}}$ due to the alphabet of $\mathcal{S}$ being a super-set of the alphabet of $\mathcal{I}$, we define a new interface $I_{N+1} = 1_{\Sigma_{\mathcal{S}} \setminus \Sigma_{\mathcal{I}}}$ and create a new complete interface $\mathcal{I}'$ such that simulation relation $R' \subseteq Q_{\mathcal{S}} \times Q_{\mathcal{I}'}$ exists. In the remainder of this paper, we assume that the simulation relation $R \subseteq Q_{\mathcal{S}} \times Q_{\mathcal{I}}$ exists (after adding a new interface if needed).

Mathematically, the decomposition problem becomes determining $\mathcal{C} = (\mathcal{R}, \mathcal{O})$ such that the following equation holds:

$$\mathcal{S} \leftrightarrows \mathcal{I} \parallel_{\mathcal{R}} \mathcal{O} \tag{1}$$

Equation (1) essentially says that the overall model, consisting of a controller and known interfaces, is bisimilar to the original specification. As noted earlier, we focus on the observer rather than the overall controller as a whole. An observer is a DLTS which, when connected to another DLTS, 'tracks' (sequences of) events performed by the DLTS. This 'connection' takes the form of parallel composition, where an observer must synchronise with *each* event of the other DLTS. Upon the occurrence of a desirable event, an observer changes state in order to 'remember' the event.

Additionally, we impose some restrictions on the observer $\mathcal{O}$: $\mathcal{O}$ is deterministic (i.e., a DLTS) and locally minimal. A locally minimal observer is essentially an observer which is recording the minimum number of words (where 'recording' implies changing state) in order to decompose the specification. If a transition

or state is removed from a locally minimal observer, the observer ceases to be an observer. We cannot guarantee finding a unique globally minimal observer, as there may exist multiple minimal observers having the same number of states (as we will see later in Section 6). Consequently, each of these observers lend differing insights into the same behaviour of the overall system, leading to different interpretations of the system.

SCT, given the specification S as a target LTS and the complete interface $\mathcal{I}$ as plants, produces an observer $\mathcal{O}$ which is almost identical to the specification $\mathcal{S}$. While the solution satisfies eq. (1), we do not consider it acceptable as the generated observer is not always locally minimal. If an observer is the same as the specification, then there is essentially no 'decomposition' being performed. While such a scenario may be unavoidable (and we show such an example), we prefer solutions where $\mathcal{O}$ is smaller than $\mathcal{S}$.

Before discussing our decomposition algorithms, we provide sketches of decompositions in order to demonstrate the kind of results we want.

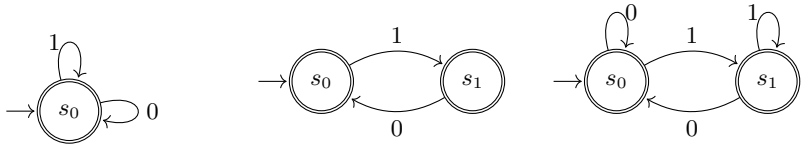
Decomposition sketch using only constraints Consider Figure 2a, showing a complete interface $\mathcal{I}$. The corresponding specification $\mathcal{S}$ is visualised in Figure 2b. In order to ensure that $\mathcal{I}$ behaves according to $\mathcal{S}$, we must constrain event *ack* at state s_0. Event *ack* can be constrained at s_0 by using the set of

(a) Unconstrained complete interface $\mathcal{I}$ (b) Specification $\mathcal{S} \cong \mathcal{I} \parallel_R 1_{\mathcal{I}}$

Fig. 2: Use of constraints: $\mathcal{S}$ is the result of constraining event *ack* at state s_0 of $\mathcal{I}$.

constraints $\mathcal{R} = \{(ack, (\mathcal{I}.s_0, \mathcal{O}.s_0))\}$ (where $\mathcal{O}.s_0$ is the only state of observer $1_{\mathcal{I}}$), obtaining $\mathcal{S} = \mathcal{I} \parallel_{\mathcal{R}} 1_{\mathcal{I}}$ (Figure 2b). Note, if there existed another transition $s_1 \xrightarrow{ack}_{\mathcal{I}} s_0$, it would not be blocked using $\mathcal{R}$.

Decomposition sketch using constraints and observer Consider the complete interface $\mathcal{I}$ in Figure 3a and its specification $\mathcal{S}$ in Figure 3b. By inspection, it is impossible to compute the set $\mathcal{R}$ such that $\mathcal{I} \parallel_{\mathcal{R}} 1_{\mathcal{I}} \cong \mathcal{S}$. As state $s_0 \in \mathcal{I}$ is being refined into two states in $\mathcal{S}$, state-based constraints alone do not suffice: $\mathcal{S}$ 'remembers' whether the previous event was 1 or 0, while $\mathcal{I}$ does not. Therefore, we need a mechanism to remember the previous event as well.
Using observer $\mathcal{O}$ (Figure 3c), it becomes possible to do so: from the initial state, if $e = 1$, then the final state is s_1, and s_0 otherwise. We can then compute $\mathcal{R} = \{(0, (\mathcal{I}.s_0, \mathcal{O}.s_0)), (1, (\mathcal{I}.s_0, \mathcal{O}.s_1))\}$. Consequently, the constrained parallel composition $\mathcal{I} \parallel_{\mathcal{R}} \mathcal{O}$ is bisimilar to $\mathcal{S}$. Observer $\mathcal{O}$ is essentially a 'flip-flop', the



(a) Complete Interface $\mathcal{I}$ (b) Specification $\mathcal{S}$ (c) Observer $\mathcal{O}$ for $\mathcal{I}$

Fig. 3: Minimal Working Example: a simple 'flip-flop' observer $\mathcal{O}$ for a flower DLTS $\mathcal{I}$

simplest type of memory element and is a globally minimal observer: it cannot be reduced any further. As such, the example presented above is a minimal example demonstrating the necessity of an observer to record historical behaviour and demonstrates a solution of the decomposition equation where the size of the observer is equal to the size of the specification.

Observers do not have to be deterministic; non-deterministic observer $\mathcal{O}'$ in Figure 4 is also usable for the decomposition in Figure 3. However, we believe

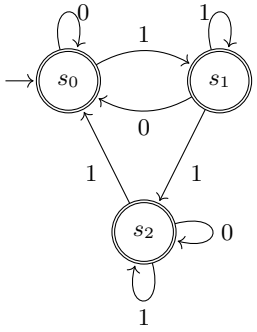


Fig. 4: Non-deterministic Observer $\mathcal{O}'$

that non-deterministic observers do not improve the understandability of the overall system and hence exclude them from our solution.

4 Related Work

In this section, we discuss some relevant work, beginning with LOTOS decomposition, continuing on to Event-B decomposition, and finally, explaining the link to Supervisory Control Theory.

Existing work on LOTOS decomposition is focused on decomposing a process given a bi-partition of events belonging to the decomposed processes. Two approaches are prominent in LOTOS: generation of a third 'control' process [6] and insertion of synthetic synchronisation events in the decomposed processes [2, 8]. The decomposition can also be performed on LTSs [7]. LOTOS decomposition

is unsuitable as we do not work with processes and addition of extra synchronisation actions is undesirable in our context. Both the aforementioned decomposition paradigms assume that the interfaces (that is, the decomposed processes) are unknown and require partitioning of the alphabet, which we do not require. Furthermore, generation of a monolithic automaton as a controller is already possible using [17].

Finally, a decomposition based on synchronising events may be *more* complex than the original model, at least by the number of states [5]. The authors of [5], however, consider the bi-partitioning of the alphabet of a monolithic model to be a simplification.

Decomposition using Event-B is focused on the topic of top-down refinement [4, 15]: given a refinement of an abstract model, a decomposition of the refined model is generated. Decomposition is achieved using (one of) two styles: shared-event or shared-variables (which encode states). Both the approaches are not applicable in our approach: we do not use common synchronising events and lack knowledge about states of interfaces by simply observing a specification. Finally, Event-B decomposition also assumes that the interfaces are unknown.

A solution for Equation (1) (the decomposition equation) can be obtained from Supervisory Control Theory (SCT) [12]. We first provide a brief explanation of SCT, followed by comparing our work to SCT. SCT solves a problem defined as follows: given a set of plants (represented as automata) and a set of requirements (represented as automata), generate a supervisor that controls the plants such that the requirements are satisfied. For simplicity, we consider a single plant automaton P and a single requirement automaton R. SCT generates a supervisor S such that the following equation holds: $S \parallel P \vDash R$, i.e., the overall behaviour of the plant and the supervisor must satisfy the specified requirement. The supervisor $S = (L, \phi)$ contains a *recogniser* L and control patterns ϕ. A control pattern is a function $\phi : Q_L \times \Sigma_L \to \{0, 1\}$, i.e., at every state of a recogniser events are either *disabled* (indicated by a 1) or *enabled* (indicated by a 0). A supervisor can be computed by composing the plants and the requirements: $S = P \parallel_{\varnothing} R$. Note, this composition satisfies the requirements $P \parallel_{\varnothing} R \vDash R$. In such a scenario, the recogniser L is equal to the automaton $P \parallel_{\varnothing} R$ and the control pattern function ϕ is defined implicitly: all events not possible in a state are automatically disabled. Figure 5 visualises a supervisor

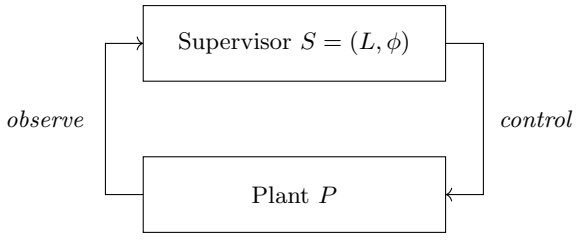


Fig. 5: Supervisory Control

and plants according to SCT. From the perspective of the supervisor, the plant is considered as a *black-box*. As the plant is a black-box, all decisions (i.e. following the control patterns) made by the supervisor must be based on the *observed* behaviour of the plant, where observations are recorded using the recogniser L.

Consequently, when SCT is applied in our case (with the complete interface as a plant, i.e. $P = \mathcal{I}$, and the specification as the requirement, i.e. $R = \mathcal{S}$), we simply obtain the specification back as a supervisor (with implicit control patterns). Furthermore, note that our task seems to be the direct opposite of SCT: SCT generates a supervisor such that the overall black-box behaviour satisfies some requirements, while we wish to extract requirements *given* the overall behaviour. It is therefore clear that our solution can be linked to SCT, but applying SCT itself does not produce a satisfactory solution (i.e. decomposing our specification).

In summary, SCT must treat the entire plant automaton as a black-box, while we treat it as a white-box, as our controller is a *part* of the system and not an external component that controls the system. The next section formalises our problem statement.

5 Decomposition Algorithms

This section discusses the methodology used to generate the decomposition discussed in Section 3. To recall, given a specification $\mathcal{S}$ and a complete interface $\mathcal{I} = I_1 \parallel_\varnothing \cdots \parallel_\varnothing I_N$, generate a controller $\mathcal{C} = (\mathcal{R}, \mathcal{O})$ such that eq. (1) ($\mathcal{S} \leftrightarrows \mathcal{I} \parallel_\mathcal{R} \mathcal{O}$) holds. Furthermore, $\mathcal{O}$ must be deterministic and locally minimal. We first discuss the conditions necessary for an observer and the procedure used to generate one. Next, we discuss how to generate constraints given a complete interface and an observer.

5.1 Decomposition observer

An observer is necessary when the *reachable* portion of a simulation relation $R \subseteq \mathcal{S} \times \mathcal{I}$ is non-injective. If the reachable simulation relation f is injective, then a flower observer ($1_\mathcal{I}$) suffices. Intuitively, if there are multiple states in $\mathcal{S}$ mapped onto a single state in $\mathcal{I}$, state-based constraints on $\mathcal{I}$ alone are insufficient *to distinguish the differences between those states in* $\mathcal{S}$, and an observer is required. An example of this type was presented earlier as a decomposition sketch in Section 3. We first explain the concept of a reachable simulation relation and then provide an algorithm to compute an observer.

Reachable Simulation Relation A simulation relation $f \subseteq Q_\mathcal{S} \times Q_\mathcal{I}$ maps each state s of the specification to a set of states in the interface which are capable of simulating state s. However, we are interested in the states of $\mathcal{I}$ which are actually *used* by $\mathcal{S}$, i.e., states which can be *reached* by using a word in the language of $\mathcal{S}$. We illustrate the concept with the help of a small example. Consider the LTSs $\mathcal{S}$ (Figure 6a) and $\mathcal{I}$ (Figure 6b) presented in Figure 6.

(a) Specification $\mathcal{S}$ (b) Interface $\mathcal{I}$

Fig. 6: Unreachable simulation: state t_1 can simulate state s_0, but can never be reached by a word ending at s_0.

The simulation relation $f \subseteq Q_{\mathcal{S}} \times Q_{\mathcal{I}}$ is $\{(s_0, t_0), (s_0, t_1), (s_1, t_0), (s_1, t_1)\}$. However, f is not a reachable simulation relation: state t_1 cannot be reached by using the empty word (since s_0 is the initial state of $\mathcal{S}$). As such, if we limit the simulation relation f to a reachable simulation relation, we obtain $f_r = \{(s_0, t_0), (s_1, t_0)\}$. However, f_r is not injective, necessitating an observer; if f_r was injective, we could begin generating constraints for $\mathcal{I}$.

An reachable simulation relation f_r can be obtained from the greatest simulation relation f by only allowing states with the same access sequence to remain in the simulation relation, i.e., $f_r = \{(q, i) \in f \mid ac(q) = ac(i)\}$ where ac is an access sequence needed to reach a state q from the initial state. We consider all the access sequences used to reach a state q as ac, with the condition that q must always be the final state and not appear as an intermediate state. If the reachable simulation relation f_r is non-injective, we generate an observer $\mathcal{O}$ for the interface $\mathcal{I}$.

Generation of Observer We begin with a definition for an observer, and then provide an algorithm to compute the same. As we currently lack the criteria for a 'good' observer, we focus on randomly generating a locally minimal observer.

Definition 6 (Observer). *An observer $\mathcal{O}$ for a specification $\mathcal{S}$ and complete interface $\mathcal{I}$ is an LTS $(Q_{\mathcal{O}}, \Sigma_{\mathcal{O}}, \rightarrow_{\mathcal{O}}, q_{\mathcal{O}})$ such that the following properties hold:*

1. *The alphabet of the observer is equal to the alphabet of the complete interface: $\Sigma_{\mathcal{O}} = \Sigma_{\mathcal{I}}$,*
2. *For each event $e \in \Sigma_{\mathcal{O}}$ and state $q \in Q_{\mathcal{O}}$, there exists a state $q' \in Q_{\mathcal{O}}$ such that $q \xrightarrow{e}_{\mathcal{O}} q'$,*
3. *There exists an injective reachable simulation relation $f_r \subseteq Q_{\mathcal{S}} \times Q_{\mathcal{I} \|_{\varnothing} \mathcal{O}}$.*

When the reachable simulation relation $f_r \subseteq Q_{\mathcal{S}} \times Q_{\mathcal{I}}$ is non-injective, historical information becomes necessary because f maps at least two states of $\mathcal{S}$ to a single state in $\mathcal{I}$ (or to the same set of states in $\mathcal{I}$). Intuitively, $\mathcal{S}$ distinguishes between (at least) two states on the basis of the traces used to reach the two states, while $\mathcal{I}$ considers the two states the same (state). An observer $\mathcal{O}$, when combined with $\mathcal{I}$, allows automaton $\mathcal{I} \|_{\varnothing} \mathcal{O}$ to distinguish the formerly indistinguishable states (see property four above).

The specification $\mathcal{S}$ can itself be used as an observer by adding self-loops at the necessary states (recall the minimal example shown in Figure 3), which we call $\mathcal{O}_{\mathcal{S}}$. Similarly, a sufficiently exploded flower can be used as an observer;

however, an exploded flower would be non-deterministic, while $\mathcal{O}_\mathcal{S}$ is an example of a deterministic observer. Consider the relation $f_s \subseteq Q_\mathcal{S} \times Q_{\mathcal{O}_\mathcal{S}}$. The relation f_s is an injective reachable simulation relation. In terms of observability, the statement "$\mathcal{S}$ is completely observable by itself" is a tautology. Using $\mathcal{O}_\mathcal{S}$ as an observer, we can compute an injective reachable simulation relation $f_r \subseteq Q_\mathcal{S} \times Q_{\mathcal{I} \parallel_\varnothing \mathcal{O}_\mathcal{S}}$. However, $\mathcal{O}_\mathcal{S}$ might not be a locally minimal observer, thus, we devise an algorithm to provide a locally minimal observer.

In order to do so, we define the exploded flower NLTS $\clubsuit_\mathcal{S}$ of specification $\mathcal{S}$. Exploded flower $\clubsuit_\mathcal{S} = (Q_\mathcal{S}, \Sigma_\mathcal{S}, (Q_\mathcal{S} \times \Sigma_\mathcal{S} \times Q_\mathcal{S}), q_s)$ is a well-connected variant of $\mathcal{S}$: each state has an edge leading to each other state for all events. With $\clubsuit_\mathcal{S}$ as an initial guess for the observer (alternatively, a *pre-observer*), we can generate a locally minimal observer by deleting as many transitions as possible (and removing unreachable states).

Conjecture 1. An observer $\mathcal{O}$ generated by Algorithm 1 is locally minimal.

Algorithm 1: `FindObserver`

Data: Pre-observer $\mathcal{O} = \clubsuit_\mathcal{S}$, Complete Interface $\mathcal{I}$, Specification $\mathcal{S}$
Result: Observer $\mathcal{O}$ such that $f_r \subseteq Q_\mathcal{S} \times Q_{\mathcal{I} \parallel_\varnothing \mathcal{O}}$ is an injective reachable simulation relation

```
1  if f_r ⊆ Q_S × Q_{I ∥_∅ O} does not exist and O ≠ ♣_S then
2  |  return ⊥ // If f is not an injective reachable simulation
   |     relation, then the current path does not generate an observer
3  if O ≅ 1_O then
4  |  return O // If O is a flower, return O
5  for E ∈ P(→_O) do
   |  // Generate sets of distinct pairs of states in O
6  |  O' := O
7  |  O := FindObserver(deleteTransitions(O, E), I, S) // deleteTransitions
   |     deletes event edges s -e->_O s' ∈ E in O, removes the unreachable
   |     states, and returns the resulting LTS
8  |  if O is a DLTS then
9  |  |  return O
10 |  else
11 |  |  O := O'
12 |  end
13 end
```

Algorithm 1 computes an observer $\mathcal{O}$. The current implementation of Algorithm 1 is not the same as presented, but the results generated by the implementation are a subset of the ones which can be generated using the above algorithm. The algorithm is recursive, proceeding in a tree-like manner, exploring possible observers by means of transition removal. The exploded flower automaton $\clubsuit_\mathcal{S}$ is provided as a pre-observer. The algorithm explores all possible

transition deletions in a depth-first manner (lines 5-12) and different observers can be generated by modifying the iteration order over all transitions on line 5. The function *deleteTransitions* deletes all transitions in the set E from the transition relation of $\mathcal{O}$ and removes any unreachable states.

If a possible path results in $f_r \subseteq S \times \mathcal{I} \parallel_{\varnothing} \mathcal{O}$ being non-injective, the path is abandoned, as it will never be possible to generate an observer from that path (lines 1-2). If an observer is reduced down to a flower, it is returned directly (lines 3-4). As there always exists a finite path of transition deletions from $\clubsuit_{\mathcal{S}}$ to $\mathcal{O}_{\mathcal{S}}$, observer $\mathcal{O}_{\mathcal{S}}$ is always returned as an observer if no other observer can be found. Algorithm 1 is guaranteed to terminate as the transition relation of $\clubsuit_{\mathcal{S}}$ is finite.

Algorithm 1 does not weigh any factors other than the size of an observer when finding an observer. This behaviour is consistent with our lack of insight into a 'better' observer, other than being locally minimal. Furthermore, Algorithm 1 can generate *all* possible observers for any decomposition problem. If necessary, the algorithm can be changed to enumerate all possible observers for a particular pair $(\mathcal{S}, \mathcal{I})$. Note that the current implementation of Algorithm 1 currently begins with $\mathcal{O}_{\mathcal{S}}$ as a pre-observer (instead of $\clubsuit_{\mathcal{S}}$) and proceeds by merging states instead of deleting transitions.

We further attempt to provide an intuition into what is a locally minimal observer. Figure 7 shows a hypothetical 'run' of Algorithm 1, returning an observer

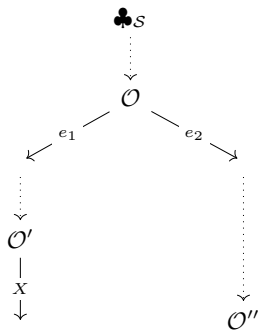


Fig. 7: Locally Minimal Observer $\mathcal{O}'$: Due to the choice of deleting event transitions E_1 as opposed to E_2, we return locally minimal $\mathcal{O}'$ as an observer, instead of a globally minimal observer $\mathcal{O}''$.

$\mathcal{O}$. Beginning with pre-observer $\clubsuit_{\mathcal{S}}$, we make a (possibly empty) set of transition deletions (indicated by the dotted arrow), reaching an observer $\mathcal{O}$, after which a choice between transition sets E_1 and E_2 is possible. We (randomly) choose E_1 and return $\mathcal{O}'$ as an observer, as any further transition deletions will not produce an observer (indicated by the 'X' over the outgoing arrow). However, if we had chosen E_2, there existed a path of transition deletions that lead to a globally minimal observer $\mathcal{O}''$. Thus, a run of Algorithm 1, due to its depth-first nature,

does not guarantee reaching a global minimum (if a unique global minimum by number of states exists). Finally, given a satisfying observer, we proceed with the generation of constraints.

5.2 Decomposition Constraints

The previous section dealt with the generation of an observer for our decomposition. In this section, we present an algorithm to generate constraints using the specification $\mathcal{S}$, complete interface $\mathcal{I}$, observer $\mathcal{O}$, and the injective reachable simulation relation $f_r \subseteq \mathcal{S} \times \mathcal{I} \parallel_{\varnothing} \mathcal{O}$. Note, if the reachable simulation relation $f'_r \subseteq \mathcal{S} \times \mathcal{I}$ is injective, then $f_r \subseteq \mathcal{S} \times \mathcal{I} \parallel_{\varnothing} \mathcal{O}$ is equivalent to $f_r \subseteq \mathcal{S} \times \mathcal{I} \parallel_{\varnothing} 1_{\mathcal{I}}$, i.e., an observer is unnecessary.

Intuitively, the injective reachable simulation relation $f_r \subseteq \mathcal{S} \times \mathcal{I} \parallel_{\varnothing} \mathcal{O}$ uniquely transforms each state in $\mathcal{S}$ to a distinct set of states in $\mathcal{I} \parallel_{\varnothing} \mathcal{O}$. For each state $s \in \mathcal{S}$, each state in the set of states $f_r(s) \in \mathcal{I} \parallel_{\varnothing} \mathcal{O}$ contains *at least* as many outgoing events as s. By blocking all outgoing events in $f_r(s)$ which are absent in s, we make $\mathcal{S}$ bisimilar to $\mathcal{I} \parallel_{\varnothing} \mathcal{O}$. This 'blocking' of events is the set of constraints $\mathcal{R}$, generated by Algorithm 2.

Algorithm 2: `FindConstraints`

Data: Specification $\mathcal{S}$, Complete Interface $\mathcal{I}$, Observer $\mathcal{O}$, Injective reachable simulation relation $f_r \subseteq \mathcal{S} \times \mathcal{I} \parallel_{\varnothing} \mathcal{O}$, projections $\pi_{\mathcal{I}} \subseteq Q_{\mathcal{I}} \times Q_{\mathcal{O}} \times Q_{\mathcal{I}}$ and $\pi_{\mathcal{O}} \subseteq Q_{\mathcal{I}} \times Q_{\mathcal{O}} \times Q_{\mathcal{O}}$

Result: Set of constraints $\mathcal{R}$

1 Create empty set B of elements of type $Q_{\mathcal{I} \parallel_{\varnothing} \mathcal{O}} \times \Sigma_{\mathcal{I}}$

2 **for** $(s, p) : f_r$ **do**

3 $\quad B(p) := \{e \in \Sigma_{\mathcal{I}} \mid t \xrightarrow{e}_{\mathcal{I} \parallel_{\varnothing} \mathcal{O}}$ *and* $s \not\xrightarrow{e}_{\mathcal{S}}\}$

4 **end**

5 $\mathcal{R} = \{(e, g(i)) \mid (e, i) \in B^{-1}\}$ where $g(i) = \{(\pi_{\mathcal{I}}(x), \pi_{\mathcal{O}}(x)) | x \in i\}$ `// Convert state` $p \in \mathcal{I} \parallel_{\varnothing} \mathcal{O}$ `to a pair of states` $(s_{\mathcal{I}}, s_{\mathcal{O}})$ `where` $s_{\mathcal{I}} \in \mathcal{I}$ `and` $s_{\mathcal{O}} \in \mathcal{O}$.

Termination of Algorithm 2 is guaranteed by the finite nature of the injective reachable simulation relation. The complexity of Algorithm 2 is $\mathcal{O}(|Q_{\mathcal{I}}| \cdot |\Sigma_{\mathcal{I}}|)$. The decomposition is formally proven in Theorem 2.

Theorem 2. *Algorithm 2 produces constraints $\mathcal{R}$ such that $\mathcal{I} \parallel_{\mathcal{R}} \mathcal{O} \leftrightarrows \mathcal{S}$.*

Proof. For simplicity, we write $Q_{\mathcal{I}\mathcal{O}}$ to indicate $Q_{\mathcal{I}} \times Q_{\mathcal{O}}$. Let $f_r \subseteq Q_{\mathcal{S}} \times Q_{\mathcal{I}\mathcal{O}}$ be the injective reachable simulation relation supplied to Algorithm 2. Note, $\Sigma_{\mathcal{S}} = \Sigma_{\mathcal{I}} = \Sigma_{\mathcal{O}}$ (by definition of complete interface and observer). The gist of our proof is thus: initially, we use the constraints $\mathcal{R}$ to compute $Y = \mathcal{I} \parallel_{\mathcal{R}} \mathcal{O}$ and show that there exists a simulation relation R from $\mathcal{S}$ to Y. In order to do so, we show that f_r needs to be injective. Next, we show that R^{-1} is also a simulation relation, proving $\mathcal{S} \leftrightarrows Y = \mathcal{I} \parallel_{\mathcal{R}} \mathcal{O}$.

Algorithm 2 generates constraints $\mathcal{R}$ by computing a set $B = \{(p, \gamma(s,p)) \mid (s,p) \in f_r\}$ where $\gamma(s,p) = \{e \in \Sigma_{\mathcal{I}} \mid p \xrightarrow{e} \textit{and } s \not\xrightarrow{e}\}$. The set B is then converted to constraints $\mathcal{R}$, used to produce $Y = \mathcal{I} \parallel_{\mathcal{R}} \mathcal{O}$. There exists a function $f : Q_{\mathcal{IO}} \to Q_Y$ mapping states from $Q_{\mathcal{IO}}$ to Q_Y, with some states in $Q_{\mathcal{IO}}$ not being mapped into Q_Y as they are now unreachable. We claim that $R = \{(s, f(p)) \mid (s,p) \in f_r\}$ is also a simulation relation. In order to do so, we first show that the reachable simulation relation f_r *must* be injective, and then prove R is a simulation relation.

We proceed with a proof by contradiction: assume that f_r is non-injective. Consider $x, y \in Q_{\mathcal{S}}$ and $z \in Q_{\mathcal{IO}}$ such that xRz and yRz. The set B then contains (at least) $(z, \gamma(x,z) \cup \gamma(y,z))$. If $\gamma(x,z) \neq \gamma(y,z)$ and we compute $\mathcal{I} \parallel_{\mathcal{R}} \mathcal{O}$, then $x \not R f(z)$ and $y \not R f(z)$ (where f is a function $f : Q_{\mathcal{I} \parallel_{\varnothing} \mathcal{O}} \to Q_{\mathcal{I} \parallel_{\mathcal{R}} \mathcal{O}}$), as some event $e \notin \gamma(x,z) \cap \gamma(y,z)$ would not be defined for state $f(z)$. Therefore R would not be a simulation relation. Hence, f_r must be injective (to see an example of this, the reader can work out the example in Figure 3 without using the observer).

Given an injective f_r, we show that $R = \{(s, f(p)) \mid (s,p) \in f_r\}$ is a simulation relation. We decompose the proof into three parts –

Initial State. Let q_s be the initial state of $\mathcal{S}$ and q_y be the initial state of Y. Observe, $q_y = f(q_p)$, where q_p is the initial state of $\mathcal{I} \parallel_{\varnothing} \mathcal{O}$ (the initial state is never unreachable). Therefore, $(q_s, q_y) \in R$ by construction.

The remaining two cases – event $e \in (\Sigma_{\mathcal{S}} \setminus \Sigma_{\mathcal{I}})$ and event $e \in (\Sigma_{\mathcal{S}} \cap \Sigma_{\mathcal{I}})$ – are treated below:

Event e $\in (\boldsymbol{\Sigma}_{\mathcal{S}} \setminus \boldsymbol{\Sigma}_{\mathcal{I}})$. As $\Sigma_{\mathcal{S}} = \Sigma_{\mathcal{I}} = \Sigma_{\mathcal{O}}$, we conclude $\Sigma_{\mathcal{S}} \setminus \Sigma_{\mathcal{I}} = \varnothing$, making this case trivially true.

Event e $\in (\boldsymbol{\Sigma}_{\mathcal{S}} \cap \boldsymbol{\Sigma}_{\mathcal{I}})$. Let $s \xrightarrow{e} s'$ and $(s,y) \in R$. We need to show that there exists a y' such that $y \xrightarrow{e} y'$ and $(s', y') \in R$. As f_r is a simulation relation, we know there exist p, p' such that $(s,p) \in f_r$ and $(s', p') \in f_r$. Furthermore, the function f sends a $p \in Q_{\mathcal{IO}}$ to its equivalent $y \in Q_Y$. Recall that all events $e \in \gamma(s,p)$ are removed when computing Y, therefore, event e is defined at state $y = f(p)$ and thus $y \xrightarrow{e} y'$. Observe that $f(p') = y'$ (since y' exists, we know that p' was mapped to its equivalent state). Hence, $(s', y') \in R$ by construction of R.

We have shown that $R = \{(s, f(p)) \mid (s,p) \in f_r\}$ is a simulation relation, we now need only show that $R^{-1} = \{(f(p), s) \mid (s, f(p)) \in R\}$ is a simulation relation in order to prove the R is a bisimulation relation. We decompose the proof into three parts again –

Initial State. As initial state pair $(s, f(p)) \in R$, true by definition.

Event e $\in (\boldsymbol{\Sigma}_{\mathcal{I}} \setminus \boldsymbol{\Sigma}_{\mathcal{S}})$. As $\Sigma_{\mathcal{S}} = \Sigma_{\mathcal{I}}$, trivially true.

Event e $\in (\boldsymbol{\Sigma}_{\mathcal{I}} \cap \boldsymbol{\Sigma}_{\mathcal{S}})$. There exists $(y = f(p), s) \in R^{-1}$ and $y \xrightarrow{e} y'$. As all events in $\gamma(s,p)$ (events at p that cannot be performed by s) have been removed using constraints $\mathcal{R}$ and $y = f(p)$, we conclude that there exists $s \xrightarrow{e} s'$. Observe, $(s,p) \in f_r$ and $(s', p') \in f_r$, therefore $(y' = f(p'), s') \in R^{-1}$ by construction.

Therefore, R^{-1} is also a simulation relation. Finally, as $R \subseteq Q_{\mathcal{S}} \times Q_Y$ is a bisimulation relation, $\mathcal{S} \leftrightarrows Y (= \mathcal{I} \parallel_{\mathcal{R}} \mathcal{O})$.

□

5.3 Complete interface to separated interfaces

The above two sections dealt with the generation of controller $\mathcal{C} = (\mathcal{R}, \mathcal{O})$ such that eq. (1) holds. Naturally, the above results need to be converted back to the separate interfaces from the complete interface.

In order to perform such a conversion, we make use of a property of parallel composition: the parallel composition $I_1 \parallel_\varnothing I_2 \parallel_\varnothing \cdots \parallel_\varnothing I_N = \mathcal{I}$ induces a projection π_i for each interface I_i for $i \in [1, N]$. Projection $\pi_i : Q_{I_1} \times Q_{I_2} \times \cdots \times Q_{I_N} \times Q_{I_i}$ maps the states of the complete interface $\mathcal{I}$ to the states of each interface. Note, a projection $\pi_i : Q_\mathcal{I} \to Q_{I_i}$ is a (functional) simulation relation.

We can then apply the projections to each constraint in $\mathcal{R}$ and transform each state in $\mathcal{I}$ to a set of states of each interface. The following section applies our approach on an example.

6 Example

This section contains a small example to show the possible decompositions that can be obtained using our approach. The example is as follows: There exist two infinitely hungry chefs, A and B who work together in a kitchen. Each prepares a dish, switches it with the other's dish, and then eats the (switched) dish. This behaviour of each chef become our interfaces. The behaviour of chef A and B is visualised in Figures 8a and 8b respectively. The kitchen $\mathcal{S}$, however, has some rules to be followed:

1. Chef A can switch her dish only after both the chefs have finished preparing their dishes,
2. Chef B can switch his dish only after chef A has switched her dish, and
3. Both chefs can eat only after chef B has switched his dish.

This is the 'externally visible' behaviour of the kitchen, which becomes the specification $\mathcal{S}$ (visualised in Figure 8c).
The parallel composition $\mathcal{I} = A \parallel_\varnothing B$ is not shown here due to space restrictions. However, $R \subseteq Q_\mathcal{S} \times Q_\mathcal{I}$ is not an injective reachable simulation relation. Thus, we generate an observer for this example.

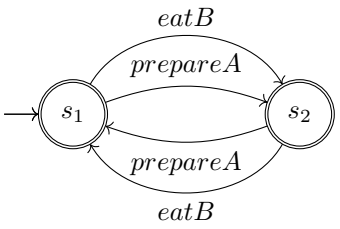


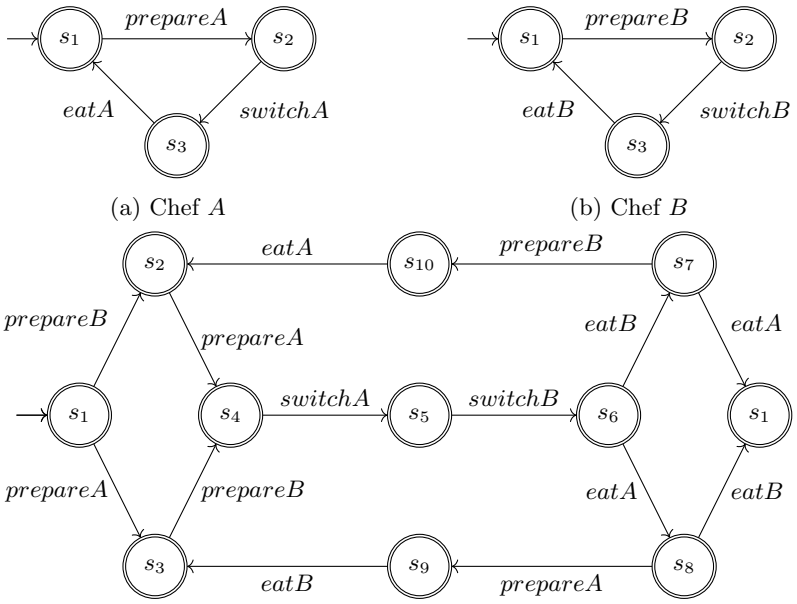
Fig. 9: Observer $\mathcal{O}_1$ (self-loops omitted)

$$R_{\mathcal{O}_1}$$

$$(eatA, (\mathcal{O}.{}_1 s_2, A.s_3, B.s_2))$$
$$(swtichB, \{(\mathcal{O}_1.s_1, A.s_3, B.s_2),$$
$$(\mathcal{O}_1.s_1, A.s_1, B.s_2),$$
$$(\mathcal{O}_1.s_2, A.s_2, B.s_2)\})$$
$$(switchA, \{(\mathcal{O}_1.s_2, A.s_2, B.s_1),$$
$$(\mathcal{O}_1.s_1, A.s_2, B.s_3)\})$$

From observer $\mathcal{O}_1$ (Figure 9) and set of constraints $\mathcal{R}_{\mathcal{O}_1}$, we can infer some behavioural rules of the kitchen. In order to simplify interpretation, we interpret[2] only the rules for event *switchA*. We obtain two rules:

[2] Kindly provided by Seray Arslan.



(a) Chef A (b) Chef B

(c) Kitchen $\mathcal{S}$. Note, $\mathcal{S}$ has only 10 states: state s_1 is duplicated for visual clarity.

Fig. 8: Infinitely Hungry Chefs

1. Event *switchA* cannot occur before chef A is prepared ($\mathcal{O}_1.s_1$) and before chef B has eaten ($B.s_3$) after switching, and
2. Event *switchA* cannot occur before chef A is prepared ($\mathcal{O}_1.s_2$) and before chef B has eaten and while chef B is prepared ($B.s_1$).

From the above two statements, one could derive the following rule: 'chef A can only switch her dish before chef B switches his dish, but she must wait until he has prepared his dish'. We now take a look at another example which leads to a different interpretation of the rule of *switchA*.

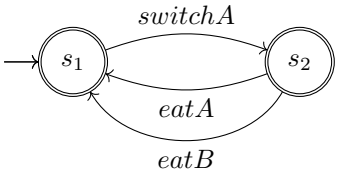


Fig. 10: Observer $\mathcal{O}_2$ (self-loops omitted)

$$
\begin{aligned}
&(eatA, (\mathcal{O}_2.s_2, A.s_3, B.s_2))\\
&(swtichB, \{(\mathcal{O}_2.s_1, A.s_1, B.s_2),\\
&\qquad (\mathcal{O}_2.s_1, A.s_2, B.s_2),\\
&\qquad (\mathcal{O}_2.s_1, A.s_3, B.s_2)\})\\
&(switchA, \{(\mathcal{O}_2.s_1, A.s_2, B.s_1),\\
&\qquad (\mathcal{O}_2.s_1, A.s_2, B.s_3)\})
\end{aligned}
$$

$R_{\mathcal{O}_2}$

Observer $\mathcal{O}_2$ (Figure 10) and set of constraints $\mathcal{R}_{\mathcal{O}_2}$ is another satisfying decomposition for our Hungry Chefs problem. We consider the constraints for event *switchA* again. Event *switchA* is 'memorised' by state s_1 of observer $\mathcal{O}_2$ and we get the following rule:

1. Event *switchA* cannot occur when chef B is not prepared ($B.s_1$) or when chef B has already switched his dish ($B.s_3$).

We could also include the state of observer $O_2.s_1$ in the interpretation, but saying '*switchA* cannot occur before *switchA* occurs' is redundant. The above interpretation of event *switchA* is clearer than the one in the previous example and possibly more intuitive to those who have seen the original rules of the kitchen.

We may fully interpret both the above observer and constraints pairs to derive the original rules again (assuming that the interface models and constraints are small enough to allow such manual reasoning), but this example shows us that different observers can lead to different interpretations of the original system. Furthermore, if we do not possess the original rules of the system, it may be impossible to derive the original rules; we must then use the rules which seem to be more 'intuitive'.

Note that both observers $\mathcal{O}_1$ and $\mathcal{O}_2$ have the same number of states: therefore defining a *global* minimum based on number of states is insufficient, we need a stricter criterion in order to define a global minimum. Finally, the state-space of $\mathcal{I} \parallel_{\varnothing} \mathcal{O}$ depends on the structure of $\mathcal{O}$; however, as events are blocked using $\mathcal{R}$, states resulting from those events are pruned away and only a structure bisimilar to $\mathcal{S}$ is preserved.

7 Conclusions

Summary of Work We have presented an approach for decomposing a deterministic system specification when its (deterministic) interfaces are known. The algorithms provided in Section 5 generate a controller consisting of state-based event constraints and an observer for a given problem. We generate a locally minimal observer to record behaviour which cannot be recorded by the state-information of the interfaces. The behaviour of the controller and interfaces is bisimilar to the behaviour of the original specification. Additionally, we have proved the correctness of our approach.

Conclusion From our work, it is clear that observers are needed when a system behaves differently at the same state at different points in time. This different behaviour is interpreted as 'memory'-based behaviour which does not exist in the interface models. Memory-based behaviour exists when the system implements rules of the 'if X occurred previously, then do Y, else do Z' sort and when the interfaces are not sufficiently refined to capture that information in their states. We capture this refinement as a locally minimal observer. Since we only guarantee locally minimal observers, this leads to non-unique interpretations of the rules followed by the overall system. Which interpretation is more 'intuitive' or 'relevant' has not been considered when conducting this work.

Threats to Validity We note that our implementation is in the proof-of-principle stage (including Algorithm 1, which has a different implementation) and we have not decomposed bigger specifications. Thus, our conclusions are not yet verified on industrial software systems and their applicability on those systems remains to be explored.

Future Work As an initial step, we may first develop a more efficient implementation of our algorithms (including Algorithm 1, which currently has a different implementation), allowing us to test our approach on bigger systems as well.

Assuming that bigger systems also generate different observers, it becomes necessary to interpret the results of the algorithms (i.e. the constraints and observer). All interpretations are considered equally valid, as they lead to the same behaviour. Thus, a natural next step would be to formalise a notion of 'relevant' or 'intuitive' for an observer, perhaps by identifying important events that need to be present in an observer and thus guiding the decomposition process.

Finally, the observer itself might be decomposed into local observers, which record behaviour per interface, per pair of interfaces, and so on, achieving a hierarchical decomposition of observers in the controller.

Bibliography

[1] Bianculli, D., Giannakopoulou, D., Pasareanu, C.S.: Interface decomposition for service compositions. In: 2011 33rd International Conference on Software Engineering (ICSE). pp. 501–510. IEEE (2011)

[2] Brinksma, E., Langerak, R.: Functionality decomposition by compositional correctness preserving transformation (1995)

[3] Cassandras, C., Lafortune, S.: Introduction to Discrete Event Systems, p. 800. Springer US (01 2010). https://doi.org/10.1007/978-0-387-68612-7

[4] Devyanin, P.N., Kulyamin, V.V., Petrenko, A.K., Khoroshilov, A.V., Shchepetkov, I.V.: Comparison of specification decomposition methods in event-b. Programming and Computer Software **42**(4), 198–205 (2016)

[5] Duhaiby, O.a., Groote, J.F.: Distribution of behaviour into parallel communicating subsystems. In: Pérez, J.A., Rot, J. (eds.) Proceedings Combined 26th International Workshop on Expressiveness in Concurrency and 16th Workshop on Structural Operational Semantics, Amsterdam, The Netherlands, 26th August 2019. Electronic Proceedings in Theoretical Computer Science, vol. 300, pp. 54–68. Open Publishing Association (2019). https://doi.org/10.4204/EPTCS.300.4

[6] Go, K., Shiratori, N.: A decomposition of a formal specification: An improved constraint-oriented method. IEEE transactions on software engineering **25**(2), 258–273 (1999)

[7] Hultström, M.: Structural decomposition. In: Protocol Specification, Testing and Verification XIV, pp. 201–216. Springer (1995)

[8] Kant, C., Higashino, T., Bochmann, G.V.: Deriving protocol specifications from service specifications written in lotos. Distrib. Comput. **10**(1), 29–47 (Jul 1996). https://doi.org/10.1007/s004460050022, `http://dx.doi.org/10.1007/s004460050022`

[9] Krohn, K., Rhodes, J.: Algebraic theory of machines. I. Prime decomposition theorem for finite semigroups and machines. Transactions of the American Mathematical Society **116**, 450–464 (1965)

[10] Moerman, J.: Learning product automata. In: International Conference on Grammatical Inference. pp. 54–66 (2019)

[11] Petrenko, A., Avellaneda, F.: Learning Communicating State Machines. In: Beyer, D., Keller, C. (eds.) Tests and Proofs. pp. 112–128. Springer International Publishing, Cham (2019)

[12] Ramadge, P.J., Wonham, W.M.: Supervisory Control of a Class of Discrete Event Processes. SIAM J. Control Optim. **25**(1), 206–230 (Jan 1987). https://doi.org/10.1137/0325013, `http://dx.doi.org/10.1137/0325013`

[13] Rath, K., Choppella, V., Johnson, S.D.: Decomposition of sequential behavior using interface specification and complementation. VLSI Design **3**(3-4), 347–358 (1995)

[14] Rath, K., Johnson, S.D.: Toward a basis for protocol specification and process decomposition. In: Computer Hardware Description Languages and their Applications, pp. 169–186. Elsevier (1993)
[15] Silva, R., Pascal, C., Hoang, T.S., Butler, M.: Decomposition tool for event-b. Software: Practice and Experience **41**(2), 199–208 (2011)
[16] Vaandrager, F.: Model learning. Commun. ACM **60**(2), 86–95 (Jan 2017). https://doi.org/10.1145/2967606, http://doi.acm.org/10.1145/2967606
[17] Yevtushenko, N., Villa, T., Brayton, R.K., Petrenko, A., Sangiovanni-Vincentelli, A.L.: Solution of parallel language equations for logic synthesis. In: Proceedings of the 2001 IEEE/ACM International Conference on Computer-aided Design. pp. 103–110. ICCAD '01, IEEE Press, Piscataway, NJ, USA (2001), http://dl.acm.org/citation.cfm?id=603095.603116