Towards Compositional Synthesis of Evolving Systems

Shiva Nejati[†] Mehrdad Sabetzadeh[†] Marsha Chechik[†] Sebastian Uchitel[‡] Pamela Zave[⋆]

[†]University of Toronto
Toronto, Canada
{shiva,mehrdad,chechik}@cs.toronto.edu

[‡]U. of Buenos Aires, Argentina &
Imperial College London, UK
suchitel@dc.uba.ar

[⋆]AT&T Labs–Research
Florham Park, NJ, USA
pamela@research.att.com

ABSTRACT

Synthesis of system configurations from a given set of features is an important and very challenging problem. This paper makes a step towards this goal by describing an efficient technique for synthesizing pipeline configurations of feature-based systems. We identify and formalize a design pattern that is commonly used in feature-based development. We show that this pattern enables *compositional* synthesis of feature arrangements. In particular, the pattern allows us to add or remove features from an existing system without having to reconfigure the system from scratch. We describe an implementation of our technique and evaluate its applicability and effectiveness using a set of telecommunication features from AT&T, arranged within the DFC architecture.

Categories and Subject Descriptors

D.2.1 [**Software Engineering**]: Requirements/Specifications

General Terms

Design, Verification

Keywords

Feature-Based Development, Synthesis, Behavioural Design Patterns, Pipelines, I/O Automata.

1. INTRODUCTION

Feature-based development has long been used as a way to provide separation of concerns, to facilitate maintenance and reuse, and to support software customization based on end-user needs [34, 24, 18, 1, 25]. Individual features typically capture specific units of functionality with direct relationships to the requirements that inspired the software system in the first place [15]. By closely mirroring requirements, features make it easier to reconfigure or expand a system as its underlying requirements change over time.

To meet the desirable properties expected from a feature-based system, the interactions among its features need to be constrained and orchestrated. This is often done by putting features in a suitable arrangement, typically a linear one such as a stack or a pipeline, that inhibits undesirable interactions.

SIGSOFT 2008/FSE-16, November 9–15, Atlanta, Georgia, USA

Existing research on feature interaction analysis, e.g., [33, 21, 19, 17, 31, 24, 14, 7], largely concentrates on reasoning about and resolving undesirable interactions between a set of features *whose arrangement is given a priori*. Yet a complementary problem, of how to automatically synthesize an arrangement when one is not given, has not been studied much. The problem is important – it currently takes substantial expertise and effort to find an arrangement of features that does not result in undesirable interactions.

Unfortunately, a naive attempt at automatically arranging features is infeasible: there is an exponential number of alternative arrangements to consider when searching for a desirable one. Hence, we need compositional techniques that can reduce the problem of finding a desirable arrangement into smaller subproblems. This need becomes even more pressing in systems that evolve over time, where features are periodically added, removed, or revised. Without compositional techniques for synthesizing evolving systems, new arrangements of features may have to be created from scratch after each change.

Our goal is to provide compositional techniques for synthesizing software systems from an *evolving*, *arbitrarily large* set of (*different*) features. To achieve this goal, we draw inspiration from the literature on component-based software. A general way to enable compositional reasoning about systems with an arbitrary number of components is by exploiting behavioural similarities between components. For example, [13, 12] show that system-wide verification tasks can be decomposed if components exhibit identical or virtually identical behaviours. The motivation for the work is verification of low-level operating system protocols, e.g., mutual exclusion where several identical copies of a process attempt to enter a critical section. More recent work, e.g., [2], explores similar ideas to bring compositional reasoning to software systems in which components have diverse behaviours. There, the required degree of similarity between components is achieved by having components implement a *design pattern*.

In this paper, we aim to study how the design patterns used in feature-based development can enable compositional synthesis of feature arrangements. We ground our work on *pipelines* – popular architectures for building feature-based systems [4, 21, 19, 24] which allow one to define the overall behaviour of a system in terms of a simple composition of the behaviours of the individual features [36].

A common objective in designing feature pipelines is to minimize the visibility of each feature to the rest. This is to ensure that individual features can operate without relying on those appearing before or after them in the pipeline [36]. To realize this objective, features are usually designed such that they engage in defining the overall behaviour of the system only when they perform their function. More precisely, features alter the flow of signals in

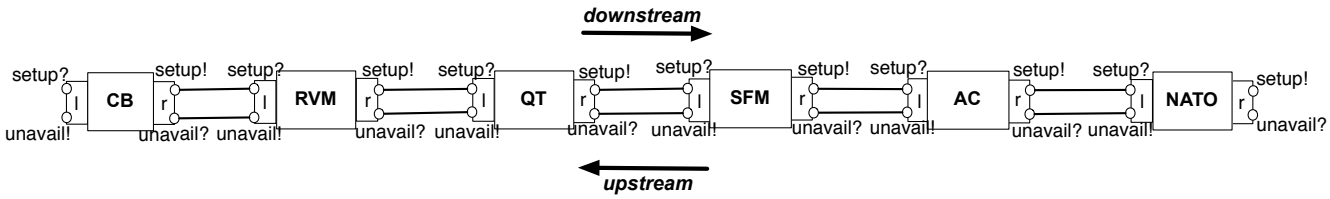


Figure 1: A simplified linear DFC scenario.

the pipeline only when they are providing their service; otherwise, they let the signals pass through without side-effects. The ability of a feature to remain unobservable to other features when it is not providing its service is called *transparency*.

In this paper, we argue that transparency is sufficient to make pipeline synthesis compositional, requiring the analysis of just pairs of features to determine their relative order in the overall pipeline. In particular, we make the following contributions:

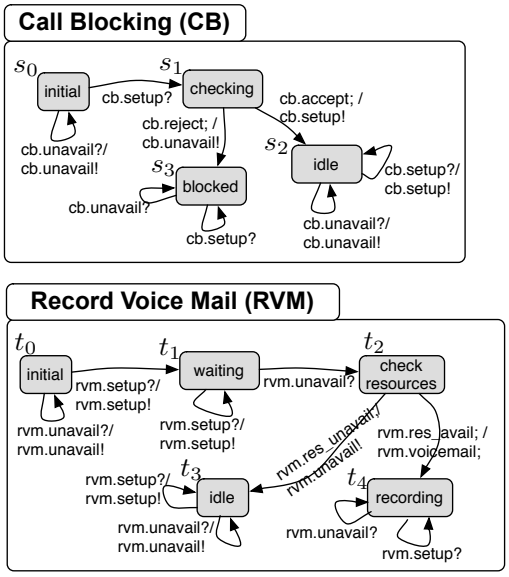
1. We formalize the transparency pattern of behaviour and show that for features implementing this pattern, *global* constraints can be inferred on the order of the features through *pairwise* analysis of the features.
2. We describe a *sound* and *complete* compositional algorithm for synthesizing pipeline arrangements. Given a set of features and a set of safety properties describing undesirable interactions, our algorithm computes an arrangement of the features that is safe for the given properties. Specifically, the algorithm uses the safety properties to compute a set of pairwise ordering constraints between the features. Due to the transparent behaviour of the features, any global ordering that violates a pairwise ordering constraint can be deemed unsafe and pruned from the search space of the solution, leaving a relatively small number of global orderings to be generated and verified by the algorithm. Our algorithm is *change-aware* in the sense that after adding or modifying a feature, we need to update only the pairwise ordering constraints related to that particular feature and reuse the remaining constraints from the previous system.
3. We report on a prototype implementation of our synthesis algorithm, applying it to a set of AT&T telecom features to find a safe arrangement for them in the Distributed Feature Composition (DFC) architecture [21]. Our algorithm could automatically and efficiently compute a safe arrangement for the DFC features in our study.

The rest of the paper is organized as follows. In Section 2, we motivate our work using an example from the telecom domain. After fixing the notation and reviewing basic notions of refinement and model checking in Section 3, we formalize features as I/O automata and define a notion of binding for describing pipelines in Section 4. Section 5 is the main contribution of the paper. It formalizes the transparency pattern that guarantees that synthesis can be done compositionally. We describe our synthesis algorithm in Section 6 and its implementation in Section 7. In Section 8, we evaluate our technique on a set of AT&T telecom features. We review related work and compare it with our approach in Section 9 and conclude the paper with a summary of contributions and an outline of future research directions in Section 10.

2. MOTIVATION

We motivate our work by analyzing a simplified instance of a telecom scenario (see Figure 1). Features in this scenario are arranged in a pipeline and include Call Blocking (CB), Record Voice Mail (RVM), Quiet Time (QT), Sequential Find Me (SFM), No Answer Time Out (NATO), and Answer Confirm (AC).

Pipeline features communicate by passing signals to their immediate neighbours. Signals that travel end-to-end pass through all features, allowing each feature to perceive and modify the overall function of the pipeline. For example, Figure 1 shows the flow of the signals setup and unavail. There are many other signal types, but we show only the most relevant ones here.



A label "e1/e2" on a transition indicates that the transition is triggered by action "e1" and generates action "e2" after being taken. Transitions can be triggered either by input actions, i.e., those received from outside, or by internal actions. When taken, a transition generates zero or more internal or output actions. It is assumed that the actions generated by a state machine do not trigger any transition of that state machine. To distinguish between input, output, and internal actions, we append to each action e the symbol "?" if e is an input action, the symbol "!" if e is an output action, and the symbol ";" if e is an internal action. Further, to disambiguate between the actions of different state machines, we prefix every action with the name of the state machine it belongs to.

Figure 2: Call Blocking (CB) and Record Voice Mail (RVM).

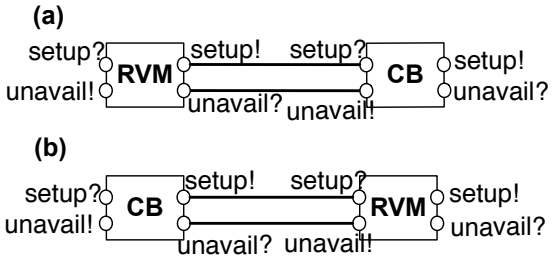


Figure 3: Possible orderings of the features in Figure 2.

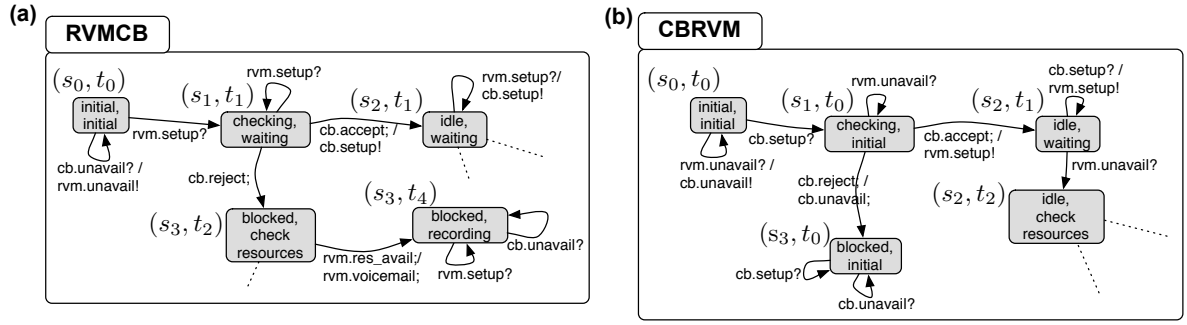


Figure 4: Fragments of the compositions of the features in Figure 2 with respect to the orderings in Figure 3.

The communication between the features in a pipeline is either buffered or unbuffered (synchronous). The former facilitates reliable communication but complicates reasoning: it is known that verification of a distributed system with unbounded buffers is undecidable [5]. Instead, we assume that features communicate through synchronous message passing, which makes for more tractable reasoning but imposes restrictions on the design of features: they should be responsive to *all* potential input at all time, i.e., they should be *input-enabled*. This requirement is captured in a number of standard formalisms for describing concurrent systems, e.g., I/O automata [26]. For example, all the features in Figure 1 are enabled for setup and unavail.

To refer to the directions within a pipeline, we use the terms *upstream* (right to left) and *downstream* (left to right). In our example, the setup signal travels downstream, and the unavail signal travels upstream. For features F and F' in a pipeline, we write $F < F'$ to indicate that F is upstream ("to the left of") of F'. For example, in Figure 1, CB $<$ NATO.

Figure 2 shows the state machines for CB and RVM in the pipeline of Figure 1. The purpose of CB is to block calling requests coming from addresses on a blocked list. CB becomes active by receiving a setup signal containing initialization data such as the directory numbers of the caller and callee. Using this data and its internal logic, CB decides whether the caller should be blocked. If so, it moves to the blocked state and tears down the call; otherwise, it moves to the idle state and effectively becomes invisible. The purpose of RVM is to record a voicemail message when the callee is not available. Like CB, RVM is activated on receipt of a setup signal. It then remains in its waiting state until it receives an unavail signal, indicating that the callee is unavailable or unable to receive the call. If the media resource is available, RVM moves to the recording state and lets the caller leave a voicemail message. Otherwise, if the media resource is unavailable, e.g., the mailbox quota for the user is exceeded, RVM moves to its idle state.

Feature Interaction. The behaviour of the composition of the features in a pipeline depends on the ordering of the features, and the goal of our work is to synthesize an ordering which will guarantee absence of undesirable compositions. For example, suppose we are trying to avoid the composition: "RVM should not record a message if CB blocks the caller" [37], formalized as the following negative trace[1]:

$$NS_1 = \mathsf{cb.reject;\, rvm.voicemail;}$$

CB and RVM can be put in a pipeline one of the two ways, as shown in Figure 3. The ordering in Figure 3(a) yields the composition in Figure 4(a), and the one in Figure 3(b) – the composition in Figure 4(b). These compositions were computed based on the parallel composition semantics in synchronous mode of communication [29]. The composition in Figure 4(a) results in an undesirable interaction: the path from (s_0, t_0) to (s_3, t_4) generates the trace NS_1, i.e., "rvm.voicemail;" comes after "cb.reject;". The composition in Figure 4(b), on the other hand, does not exhibit NS_1, implying that CB should come before RVM in a pipeline. Note that due to lack of space, Figures 4(a) and (b) only show the relevant fragments of these compositions.

[1]The trace "rvm.voicemail; cb.reject;" could have been considered instead of NS_1 as well.

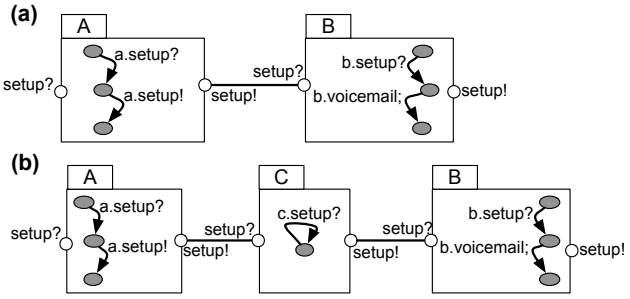


Figure 5: Local ordering vs. global ordering.

Synthesis Challenge. In general, finding a suitable ordering cannot be done compositionally when the features in a pipeline have unconstrained designs. For example, consider sample features A and B in Figure 5(a) and the property "A voicemail message should not be recorded"[2], i.e., action "b.voicemail;" must not be produced by the composition of A and B. This property does not hold over either a pipeline in which $A < B$, or the one in which $B < A$. In the former case, A sends setup to B, and B generates "b.voicemail;", and in the latter case, B receives setup from the environment and generates "b.voicemail;". So, it may seem that the given correctness property does not hold on a pipeline containing A and B. However, consider the new feature C in Figure 5(b) which blocks the action setup. The pipeline $A < C < B$ in Figure 5(b) satisfies the given correctness property, i.e., the composition of these features, when arranged in the above order, does not generate "b.voicemail;". This example shows that, in general, we may not be able to infer a global ordering over the pipeline by analyzing subsets of components. Even though the given correctness property only concerns B, our analysis needs to consider *all* the components in the pipeline. Hence, given n unrestricted components, we need to check exponentially many $(n! \approx O(2^{n \log n}))$ pipeline arrangements to find one which satisfies the properties of interest. This is intractable for all but the most trivial pipelines.

[2]This property is used only for illustration.

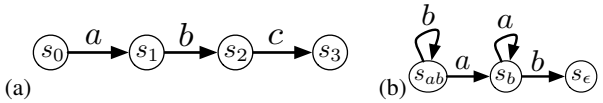


Figure 6: Examples of LTSs.

Transparency Pattern. To be able to lift an ordering over a subset of pipeline features to the entire pipeline, we rely on a pattern of behaviour called *transparency*. Each feature implementing this pattern can exhibit an execution along which it is unobservable (transparent). When executing transparently, a feature sends any signal received from its left to its right, and any signal received from its right to its left, possibly with some finite delay. Features implementing the transparency pattern can still perform their specific functionality via other executions, or via unobservable behaviours.

For example, in Figure 2, CB's transparent execution is from s_0 to s_2, and RVM's – from t_0 to t_3. CB behaves transparently if the call request comes from a non-blocked address. In this case, the system proceeds as if CB were never present; otherwise, CB provides its service by blocking the incoming call, i.e., by taking the path from s_0 to s_3. As for RVM, the feature exhibits its transparent behaviour when its media resource is unavailable; otherwise, it allows the user to leave a voicemail message by taking the path from t_0 to t_4.

For pipeline features implementing the transparency pattern, we prove the following (Section 5): if a pipeline consisting of just two features F and F' where $F < F'$ violates a safety property φ, a pipeline with an arbitrary number of components in which $F < F'$ violates φ as well. This enables a compositional algorithm for synthesizing pipeline orderings (Section 6).

3. PRELIMINARIES

In this section, we fix the notation and provide background on composition semantics, refinement, and model checking.

Labelled Transition Systems (LTS). An LTS is a tuple $M = (S, s_0, E, R)$ where S is a set of states, $s_0 \in S$ is an initial state, E is a set of actions, and $R \subseteq S \times E \times S$ is a set of transitions. We write a transition $(s, e, s') \in R$ as $s \xrightarrow{e} s'$.

Two example LTSs are shown in Figure 6. A *trace* of an LTS M is a *finite* sequence σ of actions that M can perform starting at its initial state. For example, ϵ, a, ab, and abc are traces of the LTS in Figure 6(a). The set of all traces of M is called the language of M, denoted $\mathcal{L}(M)$. We say $\sigma = e_0 e_1 \dots e_n$ is a trace over Σ if $e_i \in \Sigma$ for every $0 \leq i \leq n$. We denote by Σ^* the set of all finite traces over Σ.

Let M be an LTS, and $E' \subseteq E$. We define $M@E'$ to be the result of restricting the set of actions of M to E', i.e., replacing actions in $E \setminus E'$ with the unobservable action τ and reducing E to E'. For an LTS M with τ-labelled transitions, we consider $\mathcal{L}(M)$ to be the set of traces of M with the occurrences of τ removed. This is a standard way for hiding unobservable computations of LTSs [23].

Composition. The composition of two LTSs that run asynchronously and communicate through synchronous message passing is formalized as *parallel composition* [29]. The parallel composition operator $||$ combines the behaviours of two LTSs by synchronizing their shared actions and interleaving their non-shared ones. Unless stated otherwise, it is assumed that actions with identical names are shared, and the rest are non-shared.

DEFINITION 1 (PARALLEL COMPOSITION [29]). *Let $M_1 = (S_1, s_0, E_1, R_1)$ and $M_2 = (S_2, t_0, E_2, R_2)$ be LTSs. The* parallel composition *of M_1 and M_2, denoted $M_1||M_2$, is defined as an LTS $(S_1 \times S_2, (s_0, t_0), E_1 \cup E_2, R)$, where R is the smallest relation satisfying the following:*

$$\begin{aligned} R \;=\; & \{((s,t), e, (s',t)) \mid (s, e, s') \in R_1 \wedge e \notin E_2\} \bigcup \\ & \{((s,t), e, (s,t')) \mid (t, e, t') \in R_2 \wedge e \notin E_1\} \bigcup \\ & \{((s,t), e, (s',t')) \mid (s, e, s') \in R_1 \wedge (t, e, t') \in R_2\} \end{aligned}$$

Refinement. Refinement formalizes the relation between two LTSs at different levels of abstraction. Refinement is usually defined as a variant of *simulation*. In this paper, we use the notion of *weak simulation* (also known as observational simulation) to check the existence of a refinement relation between two LTSs [29]. This notion can be used for relating LTSs with different sets of actions by replacing their non-shared actions with τ. For states s and s' of an LTS M, we write $s \stackrel{\tau}{\Longrightarrow} s'$ to denote $s(\xrightarrow{\tau})^* s'$. For $e \neq \tau$, we write $s \stackrel{e}{\Longrightarrow} s'$ to denote $s(\stackrel{\tau}{\Longrightarrow})(\xrightarrow{e})(\stackrel{\tau}{\Longrightarrow})s'$.

DEFINITION 2 (SIMULATION [29]). *Let M_1 and M_2 be LTSs, where $E_1 = E_2 = E$. A relation $\preceq \subseteq S_1 \times S_2$ is a* weak simulation*, or* simulation *for short, where $s \preceq t$ iff*

$$\forall s' \in S_1 \cdot \forall e \in E \cup \{\tau\} \cdot s \xrightarrow{e} s' \Rightarrow \exists t' \in S_2 \cdot t \stackrel{e}{\Longrightarrow} t' \wedge s' \preceq t'$$

We say M_2 simulates M_1, written $M_1 \preceq M_2$, iff $s_0 \preceq t_0$

THEOREM 1. *[29] Let M_1 and M_2 be LTSs where $M_1 \preceq M_2$. Then, $\mathcal{L}(M_1) \subseteq \mathcal{L}(M_2)$.*

Based on the above theorem, simulation is a sufficient condition for trace containment. Recall that $\mathcal{L}(M_1)$ and $\mathcal{L}(M_2)$ capture only the *observable* behaviours of M_1 and M_2. Thus, Theorem 1 states that if $M_1 \preceq M_2$, then M_2 can generate every observable trace of M_1, but not necessarily traces with τ-steps.

Model Checking. We express correctness properties as finite negative traces over the set of actions of a system. Negative traces characterize the behaviours that a system must not exhibit (*safety* properties). For example, the property NS_1 described in Section 2 is a safety property for a telecom system with features CB and RVM. To satisfy this property, the system must not allow "rvm.voicemail;" to occur after "cb.reject;" , i.e., the trace "cb.reject; rvm.voicemail;" is a negative trace.

Let $M = (S, s_0, E, R)$ be an LTS, and let $\sigma = e_1 e_2 \dots e_n$ be a trace over E' where $E' \subseteq E$. We say that M satisfies a negative trace σ if

$$\mathit{Stut}(\sigma) \cap \mathcal{L}(M@E') = \emptyset$$

where

$$\mathit{Stut}(\sigma) = (E' \setminus e_1)^* e_1 (E' \setminus e_2)^* e_2 \dots (E' \setminus e_n)^* e_n$$

That is, the system that needs to exclude NS_1 should not allow any trace in the language $b^* a a^* b$, where $a =$ "cb.reject;" and $b =$"rvm.voicemail;", either. This can be determined by translating σ to a safety LTS M_σ and computing the parallel composition of M and M_σ (e.g., see [27]).

Formally, let σ be a trace over E'. A *safety LTS* M_σ is a tuple (S, s_σ, E', R) where

$$\begin{aligned} S =& \{s_{\sigma'} \mid \sigma' \text{ is a (possibly empty) suffix of } \sigma\} \\ R =& \{(s_{\sigma'}, e, s_{\sigma''}) \mid \sigma' = e.\sigma'' \wedge \sigma' \text{ is a suffix of } \sigma\} \cup \\ & \{(s_{\sigma'}, e', s_{\sigma'}) \mid \sigma' = e.\sigma'' \wedge e' \in E' \wedge e \neq e' \wedge \sigma' \text{ is a suffix of } \sigma\} \end{aligned}$$

For example, the LTS in Figure 6(b) can be interpreted as the safety LTS for the negative trace NS_1 in Section 2 by letting $a =$

"cb.reject;" and b ="rvm.voicemail;". Note that state s_ϵ, which corresponds to the empty suffix ϵ, is without outgoing transitions in every safety LTS. Reachability of this state determines whether M can generate σ. That is, $Stut(\sigma) \cap \mathcal{L}(M@E') = \emptyset$ iff state s_ϵ is not reachable in $M_\sigma || M$. Thus, model checking an LTS M against a negative trace σ can be done by composing M with M_σ and checking reachability of s_ϵ.

4. I/O AUTOMATA AND PIPELINES

We describe features as I/O automata [26]. This formalism is chosen because (1) I/O automata allow distinguishing between the input, internal, and output actions of features – this distinction between different types of actions is crucial for properly describing the communications between features [26]; and (2) I/O automata are input-enabled by design. Input-enabledness makes it easier to detect and avoid deadlocks [26, 38] and further, provides a way to terminate features that are stuck in error loops and hence are wasting resources [38].

DEFINITION 3 (I/O AUTOMATA [26]). *An* I/O automaton *is a tuple $A = (S, s_0, E, R)$, where S is a finite set of states; $s_0 \in S$ is an initial state; E is a set of actions partitioned into input actions (E^i), output actions (E^o), and internal actions (E^h); and $R \subseteq S \times E \times S$ is a set of transitions.*

Input actions are those that a feature receives from its environment. Internal actions represent events scoped inside a feature and invisible outside of it. Examples of such events include internal timers and communication with media devices. Output actions represent a feature's response to its input and internal actions.

An I/O automaton can be viewed as an LTS if the distinction between input, output and internal actions is ignored. Given an I/O automaton $A = (S, s_0, E = E^i \cup E^o \cup E^h, R)$, we write $LTS(A)$ to denote the LTS (S, s_0, E, R). Similar to LTSs, we write $A@E'$ to denote A with its set of actions reduced from E to E', and write $\mathcal{L}(A)$ to denote the set of traces of A. Figures 7(a) and (b) show the I/O automata for the state machines in Figures 2(a) and (b), respectively. The labels of the input and output actions of these I/O automata have infixes "r" (right) and "l" (left); these indicate the directions in which these actions are communicated (see Definition 4).

We say a state s is *enabled* for an action e if s has an outgoing transition labelled e. A state s is *quiescent* if s is not enabled for any output or internal actions. Intuitively, an automaton in a quiescent state is strictly waiting for an input from its environment. An I/O automaton A is *input-enabled* if the following conditions hold:

1. A returns promptly to some quiescent state after leaving one. We assume the execution time of transitions labelled with output and internal actions to be negligible. Thus, prompt return to a quiescent state means that output and internal actions never block the execution, and further, no cycle of transitions labelled with only internal and output actions exists.

2. Quiescent states of A are enabled for all input actions. For example, states s_0, s_3, and s_5 in Figure 7(a) are quiescent and are enabled for all input actions of CB, i.e., "cb.l.setup?" and "cb.r.unavail?".

As shown in Figure 1, each feature has one port on its left and one on its right side, and actions can be sent or received from either of these two ports. To be able to refer to the direction of communication in a pipeline, we augment I/O automata with action mappings which specify the port from which an action is sent or received.

DEFINITION 4 (FEATURES). *A* feature *F is a tuple (A_F, f) where A_F is an I/O automaton, and $f : E^i \cup E^o \to \{\mathsf{r}, \mathsf{l}\}$ is a function that maps every input and output action of F to either the right,* r*, or the left,* l*, port of F. We write "$F.\mathsf{r}.e$" (or, respectively, "$F.\mathsf{l}.e$") to say that action e is mapped to port "*r*" (or, respectively, "*l*").*

Note that f does not map the internal actions of a feature because these actions are invisible outside the feature.

In Figure 1, the smaller boxes attached to the features denote the ports. Actions are visualized as small circles on the appropriate ports. For example, CB has an (output) action "cb.r.setup!" mapped to its right port, and RVM has an (input) action "rvm.l.setup?" mapped to its left port.

To formally specify how two consecutive features in a pipeline communicate, we define a notion of *binding* for connecting the right port of one feature to the left port of another.

DEFINITION 5 (PIPELINE BINDINGS). *Let F_1 and F_2 be consecutive features in a pipeline; let $R = \{e \in E_1 \mid f_1(e) = \mathsf{r}\}$; and let $L = \{e \in E_2 \mid f_2(e) = \mathsf{l}\}$. A* (pipeline) binding *$B \subseteq R \times L$ between F_1 and F_2 is a one-to-one correspondence relation between L and R that relates input actions only to output actions, and output actions only to input actions; i.e.,*

$$(e_1, e_2) \in B \implies \big((e_1 \in E_1^o \wedge e_2 \in E_2^i) \vee (e_1 \in E_1^i \wedge e_2 \in E_2^o)\big)$$

For a binding B, we say an action is shared *if it occurs in some tuple of B, and* non-shared *otherwise.*

The links between the features in Figure 1 can be expressed as bindings. For example, the CB–RVM link in the figure is characterized by the following binding:

$$B = \{(\mathsf{cb.r.setup!}, \mathsf{rvm.l.setup?}), (\mathsf{cb.r.unavail?}, \mathsf{rvm.l.unavail!})\}$$

which indicates that the output action "cb.r.setup!" (of CB) synchronizes with the input action "rvm.l.setup?" (of RVM), and the input action "cb.r.unavail?" (of CB) synchronizes with the output action "rvm.l.unavail!" (of RVM).

In our working example, bindings are meaningful only if they relate actions with identical signal names. For example, had we considered an additional upstream-traveling signal, unknown in Figure 1, it would have been incorrect to, say, relate actions "cb.r.unknown?" and "rvm.l.unavail!". Thus, in this paper we assume that features use a unified set of signals and all bindings are based on signal name equivalences. On the other hand, we recognize that there may be domains where this assumption does not hold: features may refer to a shared signal by different names, or refer to non-shared signals by the same name. Through making mappings between actions explicit, all such bindings can be captured by Definition 5 directly.

To obtain the overall behaviour of a set of communicating features, we compose them with respect to the bindings established between them. To this end, we define a parallel composition of I/O automata, whereby features synchronize their shared actions and interleave their non-shared ones.

DEFINITION 6 (COMPOSITION OF PIPELINE FEATURES). *Let F_1 and F_2 be consecutive features in a pipeline linked by a binding B. The* parallel composition *of F_1 and F_2 with respect to B, denoted $F_1 ||_B F_2$, is a feature (A, f) where*

- *$A = (S_1 \times S_2, (s_0, t_0), E = E^i \cup E^o \cup E^h, R)$ with E^i, E^o, E^h, and R defined as follows:*

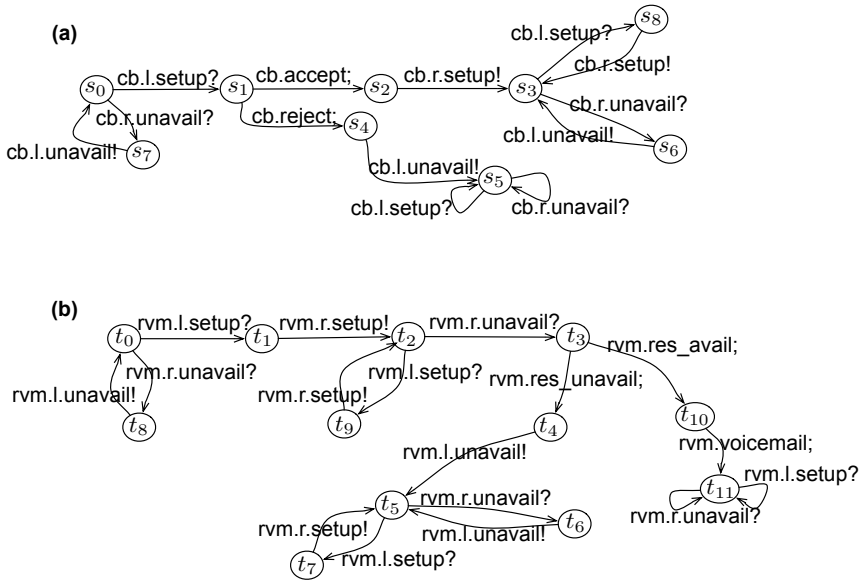


Figure 7: I/O automata for the state machines in Figure 2.

$$
\begin{aligned}
E^i &= (E_1^i \cup E_2^i) \setminus \{e \mid e \textit{ is a shared input action}\} \\
E^o &= (E_1^o \cup E_2^o) \setminus \{e \mid e \textit{ is a shared output action}\} \\
E^h &= (E_1^h \cup E_2^h) \cup B \\
R &= \{((s,t),e,(s',t)) \mid (s,e,s') \in R_1 \wedge e \textit{ is a non-shared action}\} \bigcup \\
&\quad \{((s,t),e,(s,t')) \mid (t,e,t') \in R_2 \wedge e \textit{ is a non-shared action}\} \bigcup \\
&\quad \{((s,t),(e,e'),(s',t')) \mid (s,e,s') \in R_1 \wedge (t,e',t') \in R_2 \wedge (e,e') \in B\}
\end{aligned}
$$

- $f = (f_1 \cup f_2) \setminus \{e \mid e \textit{ is a shared action}\}$

The above is the same as the standard definition of parallel composition for I/O automata [26], except that we use bindings to explicitly specify the shared actions prior to composition. Since bindings are one-to-one, it easily follows that the $||_B$ operator is associative. Thus, the global composition of the features in a pipeline can be formulated as a series of binary compositions.

5. FORMALIZING TRANSPARENCY

Intuitively, if a feature F implements the transparency pattern (motivated in Section 2), then there is *some* environment that coerces F to exhibit its transparent behaviour. For example, CB (in Figure 2) exhibits its transparent execution, i.e., from s_0 to s_2, when data from the environment indicates that the callee has not blocked the caller. Since pipeline features act independently [21, 36], each feature can be coerced into its transparent execution independently of other features.

In this section, we formalize the above intuition and prove (in Theorem 2) that if all features implement the transparency pattern, the following holds:

"If a pipeline with two features (F followed by F') violates a safety property φ, a pipeline with an arbitrary number of features in which $F < F'$ violates φ as well."

We exploit this result in Section 6 to provide a compositional algorithm for ordering features in a pipeline.

The formalization of the transparency pattern G is shown in Figure 8. It is expressed as an I/O automaton with generic input actions $G.\mathsf{l}.\langle x\rangle?$ and $G.\mathsf{r}.\langle y\rangle?$, and generic output actions $G.\mathsf{l}.\langle y\rangle!$ and $G.\mathsf{r}.\langle x\rangle!$. State S_0 is quiescent, and states S_1 and S_2 are transient. A feature implementing this pattern can exhibit some execution along which it forwards any signal it receives from its left port onto its right port, and vice versa. On this execution, a feature can delay the transmission of actions for a *finite* amount of time to perform its internal behaviours, but is not allowed to add or omit any actions, or to change the order of actions being transmitted. If the environmental data is such that a feature has to provide its service in response, the feature chooses a non-transparent execution or simply fulfills its functionality through internal actions on its transparent execution.

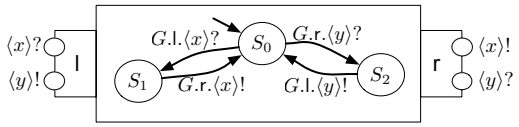


Figure 8: G: Generic transparency pattern.

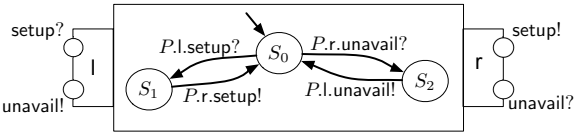


Figure 9: P: Adaptation of the generic transparency pattern to the pipeline in Figure 1.

The cycle between states S_0 and S_1 in Figure 8 (hereafter, the *downstream cycle*) handles signals that travel downstream, and the cycle between S_0 and S_2 (hereafter, the *upstream cycle*) handles signals traveling upstream. To adapt the generic transparency pattern to a specific pipeline problem, we need a copy of the downstream cycle for every signal traveling downstream, and a copy of the upstream cycle for every signal traveling upstream. For example, Figure 9 shows the adaptation, P, of the pattern to the pipeline in Figure 1. Since this pipeline has one downstream traveling signal, setup, and one upstream traveling signal, unavail, P has one copy of the downstream and one copy of the upstream cycle. Had we considered further signals, we would have had more copies of the corresponding cycles in this adaptation.

We characterize the implementation relation between a feature and its adaptation by weak simulation (see Definition 2), which allows us to relate features with different sets of actions. Having such flexibility is key: although the features in a pipeline share the same input and output actions with the pattern adaptation, each feature has its own set of internal actions. For example, consider features CB and RVM in Figure 7. CB's internal actions are "cb.reject;" and "cb.accept;", whereas RVM's are "rvm.res_unavail;", "rvm.res_avail;" and "rvm.voicemail;". Such internal actions are not used in P in Figure 9.

To establish a simulation relation between a feature and its pattern adaptation, we need to hide the feature's internal actions. For example, after replacing actions "cb.reject;" and "cb.accept;" of CB and "rvm.voicemail;" of RVM with τ, both CB and RVM simulate P. The simulation relation for CB is $\{(s_0, S_0)$, (s_1, S_1), (s_2, S_1), (s_3, S_0), (s_6, S_2), (s_7, S_2), $(s_8, S_1)\}$, and for RVM is $\{(t_0, S_0)$, (t_1, S_1), (t_2, S_0), (t_3, S_2), (t_4, S_2), (t_5, S_0), (t_6, S_2), (t_7, S_1), (t_8, S_2), $(t_9, S_1)\}$.

Before giving the main result of this section, Theorem 2, we state two lemmas used in the proof of the theorem. For the remainder of this section, let P be the adaptation of the generic transparency pattern, G, for a particular pipeline.

LEMMA 1. *Let F be a feature, and let B_1 bind F.r to P.l. If F violates a desired safety property, so does $F||_{B_1}P$. Similarly, let B_2 bind P.r to F.l. If F violates a desired safety property, so does $P||_{B_2}F$.*

The proof of this lemma follows from the fact that the set of traces of an arbitrary pipeline feature F is preserved in the composition of F with P, i.e., P does not affect traces of the composition.

The following lemma states that if the features in a pipeline implement the transparency pattern, so does the entire pipeline. That is, the features cannot prohibit one another from exhibiting their transparent behaviour.

LEMMA 2. *Let $F_1, \ldots, F_n$ be consecutive features in a pipeline, where B_i binds F_i.r to F_{i+1}.l. If every F_i ($1 \leq i \leq n$) implements (i.e., weak simulates) P, so does the composition*

$$F_1||_{B_1}F_2||_{B_2}\ldots||_{B_{n-1}}F_n.$$

Proof. We first recall two standard results on parallel composition of state transition systems (see [9]).

(1) for every M_1, M_2 and M_3, if $M_1 \preceq M_2$ then $M_3||M_1 \preceq M_3||M_2$.

(2) for every M, we have $M \preceq M||M$

The proof follows by induction on n. The base case, $n = 1$, is trivial. Let $F = F_1||_{B_1}F_2||_{B_2}\ldots||_{B_{n-2}}F_{n-1}$.

$P \preceq F_1 \wedge \ldots \wedge P \preceq F_n$
(by the inductive hypothesis)
$\Rightarrow$ $P \preceq F \wedge P \preceq F_n$
(by **(1)**)
$\Rightarrow$ $P||_{B_{n-1}}F_n \preceq F||_{B_{n-1}}F_n \wedge P||_{B_{n-1}}P \preceq P||_{B_{n-1}}F_n$
(by transitivity of $\preceq$)
$\Rightarrow$ $P||_{B_{n-1}}P \preceq F||_{B_{n-1}}F_n$
(by **(2)**)
$\Rightarrow$ $P \preceq F||_{B_{n-1}}F_n$

Note that the actions of the left and right ports of all features $F_1, \ldots, F_n$ are the same as those of P. Thus, all bindings $B_1, \ldots, B_n$ are identical. Therefore, for any B_i, the operator $||_{B_i}$ can be used to compose any pair of features or any feature with P.

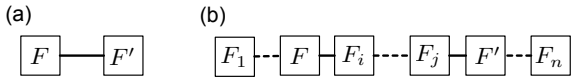


Figure 10: An illustration for Theorem 2.

Finally, we present the main theorem of this section:

THEOREM 2. *Let $F, F', F_1, \ldots, F_n$ be pipeline features, and let F, F' and every F_i ($1 \leq i \leq n$) implement P. If the pipeline in Figure 10(a) does not satisfy a desired safety property, neither does the pipeline in Figure 10(b).*

Proof (sketch). Let X_1 be the pipeline segment from F_1 to F_{i-1}, X_2 be the segment from F_i to F_j, and X_3 be the segment from F_{j+1} to F_n in Figure 10(b). Suppose X is the pipeline obtained by replacing each X_1, X_2 and X_3 in Figure 10(b) with P, i.e., X consists of F, F' and three instances of P. By Lemma 2, if X is not safe, neither is the pipeline in Figure 10(b). By Lemma 1 and Theorem 1 in Section 3, if the pipeline in Figure 10(a) is not safe, neither is X.

In Section 6, we use Theorem 2 to propose an efficient pipeline ordering algorithm. Another application of this theorem, which we do not consider in this paper, is for pipeline verification. Specifically, it follows from the contrapositive of the theorem that if a given pipeline satisfies a safety property, any subsequence of the pipeline satisfies that property as well.

6. COMPOSITIONAL SYNTHESIS

In this section, we describe the algorithm for computing ordering of features in a pipeline, to ensure that they do not admit any of the undesirable interactions. The algorithm, ORDERPIPELINE, is shown in Figure 11. The main engine of this algorithm is the function FINDPAIRWISECONSTRAINTS, shown in Figure 12, which computes a set $\mathcal{C}$ of ordering constraints between feature pairs. These constraints are inferred by model checking the two possible compositions of each feature pair against the safety properties defined over that pair. For example, let $F_1 =$ CB and $F_2 =$ RVM, and let negTr $= NS_1$ (see Section 2). With these inputs, FINDPAIRWISECONSTRAINTS yields CB $<$ RVM because the property NS_1 holds in the composition where CB comes before RVM (line 5 in Figure 12), but not in the other composition (line 7). The resulting constraint CB $<$ RVM is added to $\mathcal{C}$ (line 11) which is returned on line 16. By Theorem 2, a pipeline ordering that does not respect pairwise ordering constraints is unsafe, and thus inadmissible. This provides us with an effective strategy for pruning the search space for solutions.

Given a pair of features, FINDPAIRWISECONSTRAINTS can infer an ordering over the pair, if exactly one of their two possible compositions violates the given properties. Otherwise, if neither composition violates the properties, the features in question can be put in any order, and hence no constraint is derived (line 15). If both compositions violate the properties, FINDPAIRWISECONSTRAINTS returns error (line 9). In this case, the given features need to be revised before they can be put together in a pipeline; hence, ORDERPIPELINE terminates unsuccessfully (line 3).

If FINDPAIRWISECONSTRAINTS does not return error, ORDERPIPELINE enters a repeat-until loop (lines 4–8). Every iteration of this loop starts by finding a permutation of the n features comprising the pipeline that satisfies the set of constraints computed by FINDPAIRWISECONSTRAINTS. Such a permutation, called T, satisfies a set $\mathcal{C}$ of constraints if for every constraint $F_k < F_l$ in $\mathcal{C}$, we

Algorithm. ORDERPIPELINE
Input: - Features $F_1, \ldots, F_n$ with action sets $E_1, \ldots, E_n$, resp.
- A set $\mathsf{negTr} \subseteq (\bigcup_{1 \le k \le n} E_k)^*$ of negative traces.
Output: A permutation, T, of 1 to n giving an order on $F_1, \ldots, F_n$.

1: $\mathcal{C}$:= FINDPAIRWISECONSTRAINTS($F_1, \ldots, F_n$, negTr)
2: **if** ($\mathcal{C}$ = error) **:**
3: **return** error
4: **repeat**
5: T:= Next permutation of $1, 2, \cdots, n$ satisfying $\mathcal{C}$
6: Let B_i bind $F_{\mathsf{T}[i]}$.r to $F_{\mathsf{T}[i+1]}$.l for $1 \le i < n$
// B_i connects the feature at position i
// to the one at position $i+1$
7: safe := MODELCHECK($F_{\mathsf{T}[1]} ||_{B_1} \ldots ||_{B_{n-1}} F_{\mathsf{T}[n]}$, negTr)
8: **until** safe
9: **return** T

Figure 11: Algorithm for pipeline ordering.

Algorithm. FINDPAIRWISECONSTRAINTS
Input: Features $F_1, \ldots, F_n$ and negative trace negTr.
Output: A set $\mathcal{C}$ or pairwise ordering constraints.

1: $\mathcal{C} := \emptyset$
2: **for** $1 \le k < l \le n$**:** *// choose a pair F_k, F_l*
// restrict negTr *to F_k and F_l*
3: $\mathsf{negTr}' := \mathsf{negTr} \cap (E_k \cup E_l)^*$
4: B_1 := Binding(F_k.r, F_l.l) *// put F_k before F_l*
5: safe_1 := MODELCHECK($F_k ||_{B_1} F_l$, negTr$'$)
6: B_2 := Binding(F_l.r,F_k.l) *// put F_k after F_l*
7: safe_2 := MODELCHECK($F_k ||_{B_2} F_l$, negTr$'$)
8: **if** $(\neg\mathsf{safe}_1 \wedge \neg\mathsf{safe}_2)$ **:**
9: **return** error
10: **if** $(\mathsf{safe}_1 \wedge \neg\mathsf{safe}_2)$ **:**
11: add $F_k < F_l$ to $\mathcal{C}$
12: **else if** $(\neg\mathsf{safe}_1 \wedge \mathsf{safe}_2)$ **:**
13: add $F_l < F_k$ to $\mathcal{C}$
14: **else :** *// i.e.,* $\mathsf{safe}_1 \wedge \mathsf{safe}_2$
15: do nothing *// inconclusive result;*
// no constraint on F_k w.r.t. F_l
16: **return** $\mathcal{C}$

Figure 12: Algorithm for finding pairwise ordering constraints.

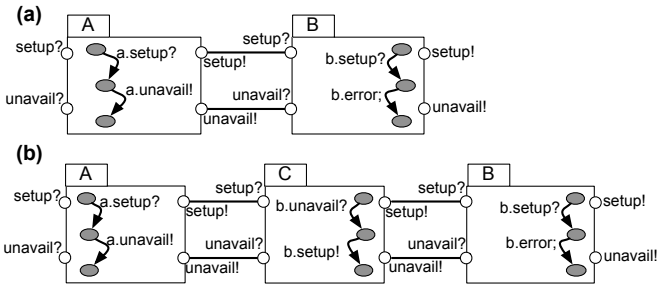


Figure 13: Local ordering vs. global ordering.

have $\mathsf{T}[k] < \mathsf{T}[l]$, i.e., feature F_k is positioned to the left of feature F_l in the pipeline. For example, let F_1 = CB, F_2 = QT, and F_3 = RVM. The permutation T satisfying constraints { CB < RVM, RVM < QT} is $[1, 3, 2]$. Afterwards, a global composition of the features is built with respect to the computed permutation T. If this composition satisfies all the given properties, T is returned as a solution. Otherwise, the loop continues until a solution is found, or all permutations that satisfy $\mathcal{C}$ are exhausted. In the latter case, ORDERPIPELINE returns error.

Notice that merely satisfying $\mathcal{C}$ does not make a given permutation T a solution to the pipeline ordering problem. For example, consider features A and B in Figure 13(a)[3]. The composition of A and B in the figure is safe for the trace "b.error;", i.e., "the error action is unreachable". However, once feature C is inserted between A and B in Figure 13(b), the resulting pipeline is no longer safe for this property: Theorem 2 only guarantees safety *violations* to lift from a pairwise to the global setting. However, safety properties that are *satisfied* over a pair of features are not necessarily lifted[4]. Therefore, we need to check all safety properties over the global composition induced by a candidate ordering. Further, although in practice most safety property traces are expressed over pairs of features, we can envision traces that refer to several and potentially to all features in the system. Checking such properties requires the construction of a global composition.

Our pipeline ordering algorithm is *sound* because we construct a global composition and verify it against all the given properties. The algorithm is *complete* because by Theorem 2, it never prunes an ordering that is a possible solution to the pipeline ordering problem. Finally, the algorithm is *change-aware*, allowing for the reuse of synthesis results across changes to pipelines. Specifically, after adding or modifying a feature F, all we need to do is to (re)compute the pairwise constraints between F and the rest of the features in the pipeline. In other words, constraints not involving F remain valid and can be carried over from the previous system.

The scalability and effectiveness of our approach ultimately depend on how well we can narrow down the search for potentially admissible pipeline permutations, and whether verifying compositions (lines 5 and 7 in Figure 12, and line 7 in Figure 11) is feasible. In Section 8, we apply our approach to an industrial telecom example. There, we demonstrate that substantial pruning of the search space can be achieved by utilizing the pairwise constraints inferred from the known undesirable interactions in the domain. The features used in our evaluation were not very large, and therefore, we could verify their compositions in a conventional way. But, for larger systems, we can improve the scalability of ORDERPIPELINE algorithm using existing automated compositional techniques for checking safety properties (e.g., [10]).

7. IMPLEMENTATION

We have developed a prototype implementation of the pipeline ordering algorithm described in Section 6. We discuss inputs to the algorithm as well as the relevant technical details below.

Inputs. Our algorithm in Section 6 receives a set of features expressed as I/O automata and a set of negative traces capturing undesirable interactions between these features. In order to use standard verification tools, in our case, the LTS Analyzer (LTSA) tool [27], our tool translates the input features to LTSs and the negative traces – to safety LTSs (see Section 3).

Parallel composition. Our technique requires us to compute compositions of pipeline features (lines 5 and 7 of FINDPAIRWISEC-

[3]This example is similar to that given in Section 2, but the details are not identical.
[4]The features in Figure 13 can be completed to implement the transparency pattern (the completions not shown here due to space limitations) and yet exhibit the same problem.

ONSTRAINTS in Figure 12 and line 7 of ORDERPIPELINE in Figure 11), for which we need to implement the parallel composition operator $||_B$ (Definition 6) – one is not readily available in LTSA. This is achieved as follows: first, we do an action relabelling to ensure that shared actions, with respect to a given binding B, have identical labels in the features to be composed. We then apply LTSA's parallel composition operator (Definition 1) to compose the features.

Model checking. Since we translate negative traces to safety LTSs, model checking (lines 5 and 7 of FINDPAIRWISECONSTRAINTS and line 7 of ORDERPIPELINE) can be done directly using LTSA. Note that our technique involves model checking not only pairwise but also the global composition (line 7 of ORDERPIPELINE). Our tool currently uses LTSA directly for this latter check, which has not presented a challenge so far because the number and the size of features we have been working with so far have been relatively small (see Section 8). However, this check may become an issue when analyzing larger systems, and in the future we intend to use an enhanced version of LTSA [10] that enables compositional model checking for safety properties. This approach applies to our work directly, since the negative traces we use are safety properties.

Ordering permutations. To generate ordering permutations that satisfy a given set of constraints (line 5 of ORDERPIPELINE), we use a backtracking constraint solver, Choco [22]. All constraints used in our approach are binary, and for those, the state-of-the-art look-ahead techniques for solving CSP problems are very efficient.

8. EVALUATION

In this section, we provide initial evidence for the usefulness of our approach through a case study from the telecom domain. Our study involves six features from AT&T deployed in the DFC architecture [21].

When conducting the study, we had a number of goals. The first goal was to check that the features present in the case study simulate our formalization of the transparency pattern in Figure 9 (**G1**). The other two goals were to investigate whether our technique can sufficiently narrow down the search for a safe pipeline ordering, which includes the ability to identify enough negative scenarios of interaction (**G2**), and to evaluate the performance of our technique on a realistic example (**G3**). We begin this section with a description of the domain of our study, and discuss the experience with the above goals in Section 8.2.

8.1 Domain Description

In DFC, a simple telecom usage is implemented by a linear pipeline such as the one shown in Figure 1. The original DFC pipeline has several additional signals, e.g., avail and unknown, which we omitted from Figure 1 for simplicity. The pipeline in the figure includes six features, namely, CB and RVM (see Section 2), as well as QT, SFM, AC, and NATO. A high-level description of the four new features, taken from [37], is as follows:

Quiet Time (QT) enables the subscriber to avoid an incoming call by activating a dialog with the caller, saying that the subscriber wishes not to be disturbed. If the caller indicates that the call is urgent, this feature allows the call to go through. Otherwise, it signals failure (unavail) upstream.

Sequential Find Me (SFM) attempts to find the callee at a sequence of locations. If the first location does not succeed, then while all the other locations are being tried, the feature plays an announcement, letting the caller know that the call is still active.

Answer Confirm (AC) uses a media resource to elicit confirmation that the call has been answered by a person rather than by a machine. If the test is not passed, it signals unavail upstream, even though the call was actually answered.

No Answer Time Out (NATO) signals failure (unavail) upstream if an incoming call is not answered after a certain amount of time.

Feature	CB	RVM	QT	SFM	NATO	AC
# of states	9	10	12	22	7	10
# of transitions	13	13	19	31	16	21

Table 1: Sizes of the resulting translations.

The DFC architecture supports dynamic architectural reconfiguration. This means that features and bindings can be created, destroyed, or reassigned at runtime. In fact, the pipeline in Figure 1 is a static snapshot of a dynamic structure. For example, in the figure, each new location tried by SFM results in a new setup signal sent downstream, and creation of new instances of AC and NATO. We do not consider such advanced capabilities here. Specifically, we abstract away feature behaviours involving runtime reconfiguration. Hence, a pipeline ordering synthesized by our technique is over a static snapshot of a (potentially) dynamic DFC pipeline. In this sense, the real value of our technique with respect to DFC is as an exploration tool through which analysts can consider different snapshots of the same pipeline and ensure that the synthesized orderings for these snapshots are consistent with one another.

The features in our case study are specified in Boxtalk [38] – a domain-specific language for specifying telecom features. Each Boxtalk specification is a state machine with a set of states and a set of transitions which can be triggered by actions. Boxtalk also provides constructs for manipulating data and media, but we do not consider these constructs in this work. Boxtalk is similar to I/O automata in that the models described in it are input-enabled; the language also distinguishes between input, output, and internal actions of features [38]. Hence, the control behaviours of Boxtalk specifications can be conveniently captured using our I/O automata-based formalism (Definition 4).

In this case study, all of the features except NATO and CB have additional ports through which they communicate with media resources that record speech, play announcements, detect touch-tones, etc. We have abstracted away from these ports, replacing their signals with internal actions such as "rvm.voicemail;". This abstraction is safe because the interaction of each feature with its media resource is independent and and logically contained within the feature, thus not affecting feature composition.

8.2 Experience

We manually translated the six Boxtalk features into I/O automata. The sizes of the translated models are shown in Table 1, whereas the original Boxtalk specifications and the resulting I/O automata are available in [30].

Our analysis indicates that all these features implement our formalization of the transparency pattern. We already exemplified the simulation relation for CB and RVM in Section 5. For the remaining features, see [30]. The realization of the transparency pattern satisfies goal **G1** and enables the application of our pipeline ordering algorithm.

G2. The scenarios used in our study are shown in Table 2 (left column). These scenarios came from [37] and from the experience of the domain expert – the last author of this paper. Note that these scenarios may not be always known in advance. To elicit them, the domain expert may have to inspect or monitor the models and their interactions using automated analysis tools. Table 2 (right column) shows the constraints inferred by our technique for the individual scenarios. These constraints were sufficient to conclusively order all the features in Figure 1 except for the SFM feature. The role of SFM is to transform a number that was dialed, i.e., a personal number, into some device number: a home phone, a cell phone, etc. Scenarios involving SFM cannot be expressed as sequences of actions because they refer to data, i.e., personal and device numbers. In this work, we do not model data and instead rely on the domain expert to provide the constraints for SFM. Specifically, CB, RVM, and QT should precede SFM because they are personal features, i.e., they apply to the personal number. In contrast, AC and NATO should follow SFM because they apply to each phone try individually, and there will be a different instance of AC and NATO for each try. Using these additional constraints, we were able to narrow down the set of possible global orderings to a single one.

While we had no problem in this domain where the nature of interactions between feature pairs was well studied and well understood, our technique may be less effective in other domains. The degree to which it narrows down the search is influenced by factors such as the size and the number of features in the domain, the amount of domain expertise available, and the existence of formal design guidelines for feature development, and all of these may vary widely.

To extend the applicability of our approach to domains where an adequate set of negative scenarios is hard to obtain, the approach can be combined with simulation and monitoring tools which assist users in identifying additional undesirable scenarios. The idea is that analysts often have certain heuristics for detecting "suspicious" behaviour, even though they may not have pinned down the exact undesirable interactions. For example, it might be dangerous for certain pairs of features to be active in the same usage scenario. The ability of a tool to report pairs of features that can be active simultaneously may help analysts to identify additional safety properties and thus reduce the number of feature orderings.

Different monitoring tools can be used in conjunction with our approach, but the one that readily integrates with our formalism is LTSA's simulation module. This module can be used to monitor the parallel composition of a set of features and report traces leading to suspicious behaviours. These traces can then be studied by analysts as potential candidates for negative scenarios. Since our approach requires traces only over pairs of features to infer ordering constraints, users can concentrate on pairwise compositions, for which traces are typically small and intuitive enough for manual inspection.

G3. We measured the time and memory performance of the different steps in our technique, applied to the features in our study. The reported times are for a PC with a 2.2GHz Pentium Core Duo CPU and 2GB of memory; our implementation used version 1.2 of Choco and version 2.3 of LTSA.

FINDPAIRWISECONSTRAINTS: Executing lines 5 and 7 of this algorithm (Figure 12) involves building pairwise compositions of LTSs and model checking them. Since I/O automata can be seen as LTSs, the sizes of our LTS translations are those shown in Table 1. The number of states of the pairwise compositions ranged between 60 to 259, and the number of transitions between 210 to 785. The running times for generating the compositions were negligible, i.e., under 1s.

Negative Scenario	Constraint(s)
QT cannot stop a caller from leaving a voicemail message.	RVM $<$ QT
A blocked caller should not be allowed to engage in a dialogue with the system (this is to avoid wasting expensive media resources).	CB $<$ AC, CB $<$ QT CB $<$ RVM
If QT is enabled and the call is not urgent, the system should not disturb the callee with a confirmation dialogue.	QT $<$ AC
The timer interval should never include the time that the system takes having a dialogue with a user (because that should not be included in the time allowance for answering).	AC $<$ NATO, QT $<$ NATO

Table 2: Negative scenarios and the resulting constraints.

To model check the compositions, we expressed safety properties as (safety) LTSs, which, for the properties in Table 2, ranged between 3 to 5 states, and 5 to 8 transitions. For example, Figure 6 can be interpreted as a safety LTS for the property NS_1 described in Section 2 by letting $a =$ "cb.reject;" and $b =$ "rvm.voicemail". The running times of individual model checking tasks were negligible.

For the six features in the study and the properties in Table 2, the total execution time of FINDPAIRWISECONSTRAINTS was 6.47s and the maximum required memory was 10M. The result of running the algorithm is the set of ordering constraints in the second column of Table 2.

ORDERPIPELINE: Line 5 of this algorithm (Figure 11) invokes a constraint solver Choco to compute a permutation satisfying the pairwise ordering constraints. The running time and memory usage of this step were negligible due to the nature of our CSP problem (see Section 7), and resulted in a single permutation that satisfied all of the pairwise constraints in Table 2.

Line 7 of the ORDERPIPELINE algorithm requires computing a global composition of the features. Since there is only one permutation satisfying the constraints in Table 2, only one global composition needed to be built and verified. The number of states and transitions in this global composition are 1.5×10^6 and 22×10^6, respectively[5]. The time and memory needed for generating this composition are 71.4s and 913M, respectively. The total model checking time, i.e., the sum of model checking times for individual properties in Table 2, was 16min, and the maximum memory requirement was 1G.

Overall, we were able to compute a safe feature ordering in about a quarter of an hour. The order that we computed is the same as the one that was produced by the domain expert via manual analysis of the feature pairs. As we discussed in Section 2, this may not be the case when the features do not satisfy the transparency pattern. The most expensive part of our algorithm is model checking of a global composition, which took about 16min. This cost is incurred no matter what approach one takes for ordering a set of features. Even if we were to select a feature ordering randomly, we would still have to build the global composition and verify it. Since the size of global compositions grows quickly, compositional techniques for dealing with space explosion are needed. While we managed to build global compositions using LTSA in our case study without resorting to compositional analysis tools, efficient tools for checking global compositions already exist and can be readily incorporated into our approach as discussed in Section 7.

[5]We have observed that global compositions for other permutations are of roughly the same size.

9. RELATED WORK

The ideas and techniques presented in this paper are related to a variety of research areas such as concurrent systems [26, 27], web service composition [16], software architectures [36, 27], etc. A complete survey of all these threads is not intended here, and instead, we focus on comparing our work with the most relevant research thrusts.

Feature interaction is a well studied problem in feature-based software development, and many approaches addressing this problem have been proposed, e.g., [21, 3, 19, 31, 17, 24]. Some of the earlier work required extensive manual intervention. For example, [3] proposed a logic-based approach for detecting undesirable interactions by manually instrumenting potential interactions with exception clauses and employing a theorem prover for finding inconsistencies. [31] developed a scheme whereby users manually specify the join points at which new features can be inserted into a base system. Feature interactions are explored by weaving the features and model checking the result.

Recent approaches offer better automation: [19] resolves undesirable interactions using predefined priorities prescribing which feature should be favoured should a conflict arise. [24] proposes a compositional method for verifying features that are composed sequentially through known interface states. [17] provides an automated unification operator for combining features (described in a functional language) with respect to given unifiers. These approaches assume that the relationships between features (i.e., priorities in [19], interfaces in [24], and unifiers in [17]) are developed *a priori*. Our work deals with a complimentary problem of how to *synthesize* these relationships.

Several approaches propose the use of architectural styles, such as layered [6, 33] or pipeline architectures [21, 4], as a way to prevent undesirable interactions. The arrangement of features within these architectures is typically done manually. Our work offers a solution to automate this task for the pipeline architectural style.

Our work also relates to [11] which focuses on verifying port-based systems (with buffered links between features). The work builds on domain-specific knowledge about the DFC architecture, such as regularity of feature properties and symmetry of communication port behaviours, to enable compositional verification of DFC usage pipelines. However, [11] does not address the synthesis of these pipelines, and also does not exploit the transparent behaviours of DFC features for reasoning.

The closest work to ours is that of [39] which reduces the effort of computing global feature orderings by partitioning features into categories, and then separately sorting the set of categories and the set of features found within each category. To make their analysis scalable, the authors use an assumption similar to transparency which is that features in a scenario can be arbitrarily added or removed at runtime. Reasoning about soundness in [39] is similar to ours; however, since the authors do not formalize transparency, they cannot reason about the completeness of their approach. In contrast, our formalization of feature behaviours and the notion of transparency enables us to show completeness in addition to soundness.

Compositional analysis extends the applicability of verification methods by reducing reasoning about a large system to reasoning about its individual components. For example, assume-guarantee reasoning [32] enables verification of individual components in conjunction with assumptions about their environment, and allows lifting of the results to the entire system. Existing work on this topic is not directly applicable to synthesis of feature-based systems because one needs to know about the bindings between features in order to specify the environmental assumptions. However, our synthesis algorithms could directly benefit from *automated* assume-guarantee techniques for safety properties, e.g., [10]. Specifically, our feature ordering algorithm in Section 6 involves model checking pairwise and global compositions induced by specific bindings. These model checking tasks can be done more efficiently using the technique of [10].

Design for verification promotes the use of domain-specific patterns and guidelines to facilitate efficient automated verification [35, 28, 2, 8]. For example, [2] provides a concurrency controller pattern for making verification of concurrent Java programs more scalable, and [35] studies the use of structural design guidelines for efficient verification of UML models. To the best of our knowledge, the use of patterns for compositional reasoning about feature-based systems has not been investigated previously.

10. CONCLUSION

In this paper, we presented a sound and complete compositional approach for synthesizing feature arrangements. The formal groundwork for our technique is a pattern of behaviour called transparency. We proved that this pattern enables inferring global constraints on feature arrangements through pairwise analysis of the features. We also reported on a prototype implementation and a preliminary evaluation of our work for synthesizing orderings of AT&T telecom features.

In our case study, the desired properties were described as negative traces (safety). Our algorithm can also readily work with positive traces. In fact, a corollary of Theorem 2 is that the transparency pattern lifts existence of positive traces from a pairwise to the global setting. We leave extending our technique to (finite) liveness, i.e., dealing with *universal* positive behaviours, to future work.

Further, the properties in our case study did not contradict one another. However, one could envision cases where some of these properties are mutually inconsistent. Our feature ordering algorithm detects such inconsistencies as circular, or over-constrained, orderings. Alternatively, users may want to establish consistency before applying our algorithm, e.g., using Alloy [20].

While this paper focused on pipeline arrangements, our technique can be used for synthesis of more complex arrangements as well. In particular, we can synthesize graph arrangements consisting of linear pipeline segments by first synthesizing the segments and then combining them to construct the overall system.

Our current approach assumes that each feature implements a distinct requirement. Yet, it is possible for individual features, particularly in legacy systems, to implement multiple functions, each invoked depending on external parameters or user preferences. Our approach may create circular dependencies between such features. We leave the methodology of breaking these cycles and dealing with parameterized feature arrangements for future work.

Acknowledgments. We would like to thank members of the IFIP Working Group 2.9 Software Requirements Engineering, and in particular Jo Atlee, for tremendously valuable feedback on this work. We are grateful to the anonymous FSE reviewers for their useful comments. Financial support was provided by OGS, NSERC, CERAS, EPSRC, CONICET, and ANPCyT.

11. REFERENCES

[1] D. Batory. "Feature-Oriented Programming and the AHEAD Tool Suite". In *Proceedings of ICSE'04*, pages 702–703, 2004.

[2] A. Betin-Can, T. Bultan, M. Lindvall, B. Lux, and S. Topp. "Application of Design for Verification with Concurrency Controllers to Air Traffic Control Software". In *Proceedings of ASE'05*, pages 14–23, 2005.

[3] J. Blom, B. Jonsson, and L. Kempe. "Using Temporal Logic for Modular Specification of Telephone Services". In *Proceedings of FIW'94*, pages 197–216, 1994.

[4] K. Braithwaite and J. Atlee. "Towards Automated Detection of Feature Interactions". In *Proceedings of FIW'94*, pages 36–59, 1994.

[5] D. Brand and P. Zafiropulo. "On Communicating Finite-State Machines". *J. ACM*, 30(2):323–342, 1983.

[6] R. Brooks. "A Robust Layered Control System for a Mobile Robot". *IEEE J. of Robotics and Automation*, pages 2–27, 1986.

[7] G. Bruns. "Foundations for Features". In *Proceedings of FIW'05*, pages 3–11, 2005.

[8] B. Cheng, R. Stephenson, and B. Berenbach. "Lessons Learned from Automated Analysis of Industrial UML Class Models (An Experience Report)". In *Proceedings of MoDELS'05*, pages 324–338, 2005.

[9] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.

[10] J. Cobleigh, D. Giannakopoulou, and C. Pasareanu. "Learning Assumptions for Compositional Verification". In *Proceedings of TACAS'03*, pages 331–346, 2003.

[11] A. J. Dominguez and N. Day. "Compositional Reasoning for Port-Based Distributed Systems". In *Proceedings of ASE'05*, pages 376–379, 2005.

[12] A. Emerson and V. Kahlon. "Reducing Model Checking of the Many to the Few". In *Proceedings of CADE'00*, pages 236–254, 2000.

[13] A. Emerson and K. Namjoshi. "Reasoning about Rings". In *Proceedings of POPL'95*, pages 85–94, 1995.

[14] A. Felty and K. Namjoshi. "Feature Specification and Automated Conflict Detection". *ACM TOSEM*, 12(1):3–27, 2003.

[15] K. Fisler and S. Krishnamurthi. "Decomposing Verification by Features". In *Proceedings of VSTTE'05*, 2005.

[16] X. Fu, T. Bultan, and J. Su. "Analysis of Interacting BPEL Web Services". In *Proceedings of WWW'04*, pages 621–630, 2004.

[17] R. Hall. "Feature Combination and Interaction Detection via Foreground/Background Models". *Computer Networks*, 32(4):449–469, 2000.

[18] W. Harrison, H. Ossher, P. Tarr, V. Kruskal, and F. Tip. "CAT: A toolkit for assembling concerns". Technical Report RC22686, IBM Research, 2002.

[19] J. Hay and J. Atlee. "Composing Features and Resolving Interactions". In *Proceedings of FSE'00*, pages 110–119, 2000.

[20] D. Jackson. *Software Abstractions Logic, Language, and Analysis*. The MIT Press, 2006.

[21] M. Jackson and P. Zave. "Distributed Feature Composition: a Virtual Architecture for Telecommunications Services". *IEEE TSE*, 24(10):831 –847, 1998.

[22] F. Laburthe and N. Jussien. "The Choco Constraint Programming System". http://choco-solver.net/.

[23] K. Larsen, B. Steffen, and C. Weise. "A Constraint Oriented Proof Methodology based on Modal Transition Systems". In *Proceedings of TACAS'95*, LNCS, pages 13–28. Springer, May 1995.

[24] H. Li, S. Krishnamurthi, and K. Fisler. "Verifying Cross-cutting Features as Open Systems". In *Proceedings of FSE'02*, pages 89–98, 2002.

[25] H. Li, S. Krishnamurthi, and K. Fisler. "Modular Verification of Open Features Using Three-Valued Model Checking". *J. of Automated Softw. Eng.*, 12(3):349–382, 2005.

[26] N. Lynch and M. Tuttle. "Hierarchical Correctness Proofs for Distributed Algorithms". In *Proceedings of PODC'87*, pages 137–151, 1987.

[27] J. Magee and J. Kramer. *"Concurrency: State models and Java Programming: 2nd Edition"*. Wiley, 2006.

[28] P. Mehlitz and J. Penix. "Design for Verification with Dynamic Assertions". In *Proceedings of SEW'05*, pages 285–292, 2005.

[29] R. Milner. *"Communication and Concurrency"*. Prentice-Hall, New York, 1989.

[30] S. Nejati. "Translating BoxTalk Models to I/O Automata", 2007. `http://www.cs.toronto.edu/~shiva/synthesis/`.

[31] M. Plath and M. Ryan. "Feature Integration Using a Feature Construct". *Science of Computer Programming*, 41(1):53–84, 2001.

[32] A. Pnueli. "In Transition from Global to Modular Temporal Reasoning about Programs". *Logic and Models of Concurrent Systems*, pages 123–144, 1985.

[33] K. Pomakis and J. Atlee. "Reachability Analysis of Feature Interactions: A Progress Report". In *Proceedings of ISSTA'96*, pages 216–223, 1996.

[34] C. Prehofer. "Feature-Oriented Programming: A Fresh Look at Objects". In *Proceedings of ECOOP'97*, pages 419–443, 1997.

[35] N. Sharygina, J. Browne, and R. P. Kurshan. "A Formal Object-Oriented Analysis for Software Reliability: Design for Verification". In *Proceedings of FASE'01*, pages 318–332, 2001.

[36] M. Shaw and D. Garlan. *"Software Architecture: Perspectives on an Emerging Discipline"*. Prentice-Hall, Inc., 1996.

[37] P. Zave. "FAQ Sheet on Feature Interaction". `http://www.research.att.com/~pamela/faq.html`.

[38] P. Zave and M. Jackson. "A Call Abstraction for Component Coordination". *Elect. Notes in Theor. CS*, 66(4), 2002.

[39] A. Zimmer. *"Prioritizing Features Through Categorization: An Approach to Resolving Feature Interactions"*. PhD thesis, University of Waterloo, 2007.