Feature-Oriented Modelling in BIP: A Case Study

Cecylia Bocovich Joanne Atlee
University of Waterloo
Email: {cbocovic, jmatlee}@uwaterloo.ca

***Abstract*—In this paper, we investigate the usage of Behaviour-Interaction-Priority version 2 (BIP2), a component-based modelling framework, for specifying feature-oriented systems. We evaluate BIP2 in the context of the Feature Interaction Problem and quantify the amount of work needed to add features to an existing system (i.e., in terms of rework to existing features, and work to identify and specify interactions). We present the results of a case study on a telephony system with five optional features where we found that the amount of work depends heavily on how features are interconnected. We identify a number of different strategies for interconnecting features, and propose one that reduces the amount of work and rework needed to add new features to an existing system.**

I. Introduction

In software engineering, an increasingly popular strategy to decompose a complex system into smaller subproblems is to perform **feature-based decomposition**, which is a type of functional decomposition of the system. A **feature** is a unit of functionality that can be developed and evolved independently.

Although the practice of treating features as separate concerns eases the task of feature development, complexities arise in the integration of features into a final product. The composition of separately designed features often leads to unexpected or undesirable behaviours. A **feature interaction** occurs whenever one feature alters the behaviour of another. For example, a user may subscribe to a telephony feature that automatically forwards calls that are received when she is on the phone; she may also subscribe to a second feature that automatically screens calls against a list of blocked numbers. If each feature is specified and developed without knowledge or consideration of the other feature, the outcome is not clear if both are activated in the same scenario.

To be safe, a developer must consider how a new feature might interact with existing features. To be thorough, all combinations of existing features need to be considered, as in some cases an interaction occurs only when multiple features are active at the same time. As the number of features grows, the number of feature combinations that must be analyzed for possible interactions grow exponentially – until the integration of a new feature is dominated by the analysis and resolution of these feature interactions. In systems with high variability, the **Feature Interaction Problem** becomes intractable with existing methods [1].

Many techniques and tools have been developed to address the Feature Interaction Problem by minimizing the work of the developer in discovering and resolving feature interactions [2]. One such strategy is the use of specialized modelling languages for the design and verification of composed systems. **Behaviour-Interaction-Priority version 2 (BIP2)** [3], [4] is a framework for the design of component-based systems. BIP allows the designer to decompose a complex system into a collection of interconnected components, each describing orthogonal and independently executing **behaviour**. A component can modify other components through explicitly defined communications and synchronizations called **interactions**, which may cause transitions within the affected components or updates to the values of component variables. Nondeterminism and conflicts among concurrent transitions and variable assignments can be resolved using **priorities**.

Given that the BIP2 formalism is designed to support component-based modularity, and given that BIP2 has explicit language constructs for specifying how feature combinations ought to synchronize and how conflicts and nondeterminism ought to be resolved, we set out to investigate how to use BIP2 to address the Feature Interaction Problem. We performed a case study in which we use BIP2 to model a telephony system with five features. Telephony is an example of an extreme product line, with many optional features that extend the functionality of a shared basic service. Each user subscribes to a subset of available features, and establishes voice connections with one or more other users, each of who in turn has subscribed to her own subset of available features.

We aim to answer the following questions in our investigation: (1) Is it possible to model features independently and integrate them into the system without changing existing features? (2) How much work (and rework) is required to integrate a new feature into an existing system model? (3) How much work is required to specify interactions among features, and what is the overall complexity of the resulting system model? Is feature-oriented specification in BIP2 feasible? Answers to these questions depend heavily on the design methodology used to define component interfaces and to interconnect components. We identify three distinct design methodologies for modularizing and composing features, and we evaluate the amount of developer work that is needed to integrate new features and resolve feature interactions.

Given how the term *interaction* is overloaded, we want to distinguish between traditional *feature interactions* (which refer to any difference in feature behaviour, intended or not, due to the presence of other features) from *BIP2 interactions* (which refer to explicitly specified communications and synchronizations among connected components, and are by definition designed and desired interactions). We use the acronym **FIs** to refer to the former and the qualified term **interactions** to refer to the later.

II. Overview of BIP

Behaviour-Interaction-Priority (BIP) is a component-based language for modelling complex systems [3]. In BIP, the behaviour of a system is modelled as a collection of individual components, each of which is responsible for a subset of the system's behaviour. As the name suggests, BIP provides three layers of specification to the model: the Behaviour of system components, the Interactions between these components, and the Priorities between multiple possible execution paths. In this paper, we use the second iteration of this framework, BIP2 [4], and will refer to this version from this point forward. In this section, we describe each layer of BIP2, the specification of models in the BIP2 language, and how these layers are combined to form a model of a simple system.

A. The Behaviour Layer

Each component in a BIP2 model defines a subset of a system's overall functionality. In this paper, our system consists of a base component that provides basic call-processing functionality (i.e. on-demand voice connections between two users), and a set of optional features that extend or override this functionality.

The most basic BIP2 component is an **atom**. The internal operation of each atom is modelled as a Petri net. A transition between places in the net updates the atom's variables and the set of occupied places. Transitions may be optionally triggered by communications received through the atom's ports.

An atom's current state is represented by the set of currently occupied places and the values of the atom's variables. A transition from a set of previously occupied places to a set of newly occupied places may be optionally labelled with a guard, an update function, and a port. A guard is a predicate over the atom's variables, and a transition is enabled and executed only if the system state satisfies the guard. After transitioning, the variables are updated as dictated by the update function. Ports facilitate the synchronization of components' transitions, and are used in the specification of the interaction layer.

Figure 1 shows places and transitions from a partial atom that models the basic-call service — that is, a service that allows a user to to place and answer phone calls. The places of the atom reflect the states through which a telephone call can proceed, from IDLE (i.e. waiting for a call) to CALLING, to a fully connected call. The integer variables *rings* and *time* keep track of the number of successive rings and the duration of the call, respectively.

When a user initiates a call, her basic-call service eventually enters the CALLING state and remains there until the call is answered or until the call has rung a maximum number of times. The *call* transition has a guard function [rings $<$ maxRings] that ensures the transition will execute only if the maximum number of rings has not been reached; each execution of the transition increments the $rings$ variable through an update function {rings++}.

Note that every transition in this model has an additional label that is neither a guard nor an update function. These

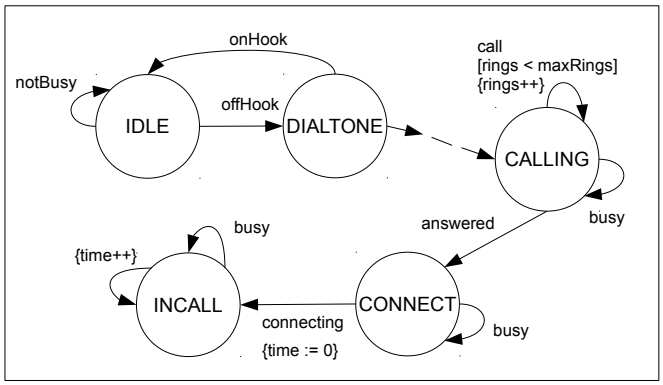


Fig. 1: Places and transitions in the basic-call service component, simplified for brevity. Guard functions are given in square brackets (e.g., $[rings < maxRings]$) and update functions in curly brackets (e.g., $\{rings++\}$).

labels correspond to ports and reflect the ways in which other components affect the internal state. Ports restrict transitions similar to guards; a transition labelled by a port relies on an interaction with another component to execute. We will discuss ports and interactions between components in greater detail in the following section. We give the full atomic component for the basic-call service in Figure 2.

B. The Interaction Layer

Components interface with each other through ports that are linked together by **connectors**. A connector links two or more components: the effect is to synchronize transitions that are labelled with connected port names and to update variables through **interactions**. Each connector explicitly defines a set of ports that belong to participating components and the set of interactions that may occur between these components.

As stated in the previous section, ports may label one or more transitions in a component. A port is enabled when one or more of these transitions are enabled. The ports in a connector may be either triggering ports (i.e. senders) or synchronizing ports (i.e. receivers). A triggering port, denoted by a primed port name (e.g. busy'), triggers transitions in the connected components when it is enabled. A synchronizing port, when enabled, allows its transitions to be triggered by a triggering port from another connected component.

Each interaction in a connector consists of the connector's triggering ports and some subset of synchronizing ports. Interactions may be labelled with guard and transfer functions in the same manner as component transitions. The variables in these functions are the data variables exported by the components' ports. Upon execution, the interaction's transfer function updates the variables in participant atoms, allowing components to exchange information.

An interaction a with a set T of triggering ports and a set S of synchronizing ports is enabled if and only if all of the following conditions are met: (1) there is at least one enabled transition labelled by each port $t' \in T$, (2) there is at least

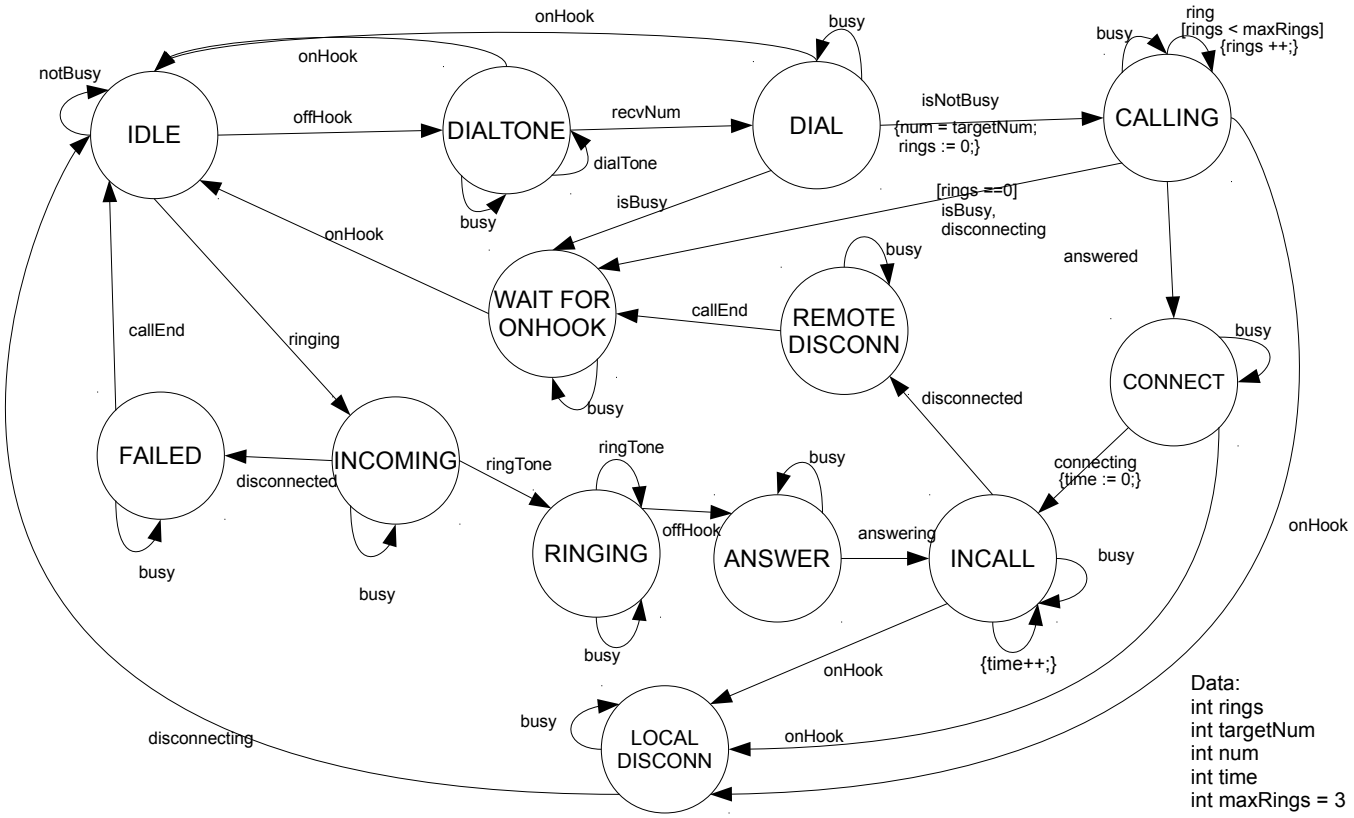


Fig. 2: A complete basic-call service component with all ports, guards, update functions, and data variables.

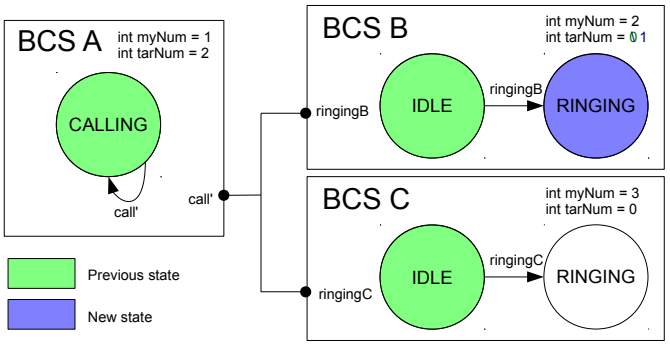


Fig. 3: Example connector between three basic-call services, simplified for brevity. The connector allows the synchronization between the three components, transitioning in a single execution step from the state shown in green to a new state shown in blue.

one enabled transition labelled by each port $s \in S$, and (3) the guard on interaction a evaluates to *true* given the current values of all exported variables.

A connector with n synchronizing ports has 2^n possible interactions. The modeller must explicitly define guards and update functions for *all* possible interactions. Any interactions that she does not wish to execute may be given the guard $G_a : false$. If more than one of a connector's interactions are enabled, the *maximal* interaction (i.e. the interaction with the most ports) will execute. For example, if there are two enabled interactions in a connector C, and one includes ports $\{t',\ s_1\}$ while the other includes ports $\{t',\ s_3,\ s_4,\ s_5\}$, the latter interaction will execute as it contains the higher number of ports. This set of implicit priority rules is also known as the *maximal progress principle*.

After an interaction occurs, the variables exported through the participating ports are updated as defined by the transfer function. In this way, an enabled interaction changes the internal state of participant components by (1) executing transitions labelled with the ports in the interaction and (2) updating the values of connected component variables.

Figure 3 shows a connector among the basic-call services of three users that enables user A to place a call to user B or user C[1]. The ports involved in the connector are the *call'* triggering port from BCS A and the *ringing* synchronizing ports of BSC B and BCS C. Each ports exports two integers: the $myNum$ variable that stores the user's unique telephone number, and the $tarNum$ variable that identifies the telephone number of a remote basic-call service. There are four possible interactions within this connector:

- $\{call',\ ringingB,\ ringingC\}$ or $\{call'\}$: During a normal call, user A calls the unique phone number of one other user, causing only that user's phone to ring,

[1]analogous connectors are needed to enable users B and C to place calls.

not both. We add a guard on these interactions to prevent this interaction from occurring: $G_a : false$.

- $\{call', ringingB\}$ or $\{call', ringingC\}$: The first interaction occurs if and only if the number that user A calls matches the phone number in $ringingB$'s component: $G_a : call.tarNum == ringingB.myNum$. If this interaction is enabled, BCS B's $tarNum$ must be updated to reflect the fact that it is now interacting with BCS A. This is done with the transfer function $F_a : ringingB.tarNum := call.myNum$. We use analogous guards and transfer functions for an interaction with user C.

We give a full interaction model of our basic call service in Figure 4. The connectors between basic-call services show the ways in which users' services may interact throughout the process of a call. Likewise, the connectors between a user component and its basic-call service model how a user may interact with the system. In this manner, the modeller can use connectors, interactions, guards, and transfer functions to control the ways in which components synchronize with each other and modify each others' data variables.

C. Priorities

To combat nondeterminism and enforce scheduling policies, BIP2 provides **priorities** as a means to choose between multiple enabled execution paths. Nondeterminism arises when there are multiple simultaneously enabled interactions, each leading to a different overall system state. In Figure 1, when the CALLING place is occupied, all transitions leading from CALLING are enabled. This potentially enables interactions in the connectors that connect the basic-call service through the $busy$, $call$, and $answered$ ports to the user component and the call-service components of other users. As discussed in the previous section, BIP2 will execute the maximal enabled interaction *within* each connector, but if there is more than one connector, there are no guarantees about which interaction will execute. In this case, we want the $answered$ signal to have the highest priority; when another user answers the call, the basic-call service should progress to the CONNECT state instead of continuing to call the other user's BCS through the $call$ port. We can control the outcome by specifying priorities in one of two ways: (1) at the component level by specifying that port $answered$ has a higher priority than port $call$, or (2) at the interaction level by specifying that interactions in the $answered$ connector have priority over interactions in the $call$ connector.

At the component level, we include $onHook > busy$ in our specification of the basic-call service atom. At the interaction level, we denote the connector that connects $onHook$ with the user atom as $onHook\_connector$ and the connector that connects the $busy$ port with other basic-call services as the $busy\_connector$. The priority $onHook\_connector : * > busy\_connector : *$ indicates that all interactions in $onHook\_connector$ should execute before interactions in $busy\_connector$.

The simplest way to resolve all nondeterminism is to define a complete ordering on the transitions that lead from each state. Note that not all pairs of transitions with the same initial state require a priority. In the DIAL state, $isNotBusy$ and $isBusy$ will not be enabled simultaneously. This would require the target basic-call service to be in two places at once: IDLE and in one of the places that are involved in a call. Our basic-call service atom requires a total of 26 priorities to resolve conflicts from simultaneously enabled interactions and avoid inconsistent states. We will see priorities play a large role in the resolution of feature interactions in Section IV.

III. Telephony Case Study

We conducted a case study on a telephony system to assess the extent to which BIP2 combats the Feature Interaction Problem by reducing the work a developer must perform when designing, composing, and resolving interactions between features. In this section, we outline the basic structure of our telephony system, the features involved, and the criteria we used to evaluate design strategies in BIP2.

A feature-oriented BIP2 model consists of three parts: (1) a **base service** (modelled as one or more atomic components), (2) a set of optional **features** to which a user may subscribe that extend or modify the functionality of the base service (each of which is modelled as an atomic component), and (3) the system **environment** (modelled as one or more atomic components). A telephony model consists of a basic-call service, a set of optional features, and a user environment. Each user has a unique phone number. A user A may dial the unique number of another user B in order to initiate a call (i.e. voice connection) to user B.

Each user subscribes to a basic-call service (BCS). This service allows the user to place and receive calls; the places in the BCS component, together with its variables, reflect the possible states of an outgoing or incoming call. The ports of the component reflect the ways in which users and features may interact with or extend the functionality of the BCS (i.e. taking the phone off the hook, or dialing a number), and the ways in which the BCS of one user interacts with the BCSs of other users (i.e. establishing a connection).

A BIP2 model is a **closed-world model**, meaning that it includes a representation of the system's operating environment. The environment of a telephony system comprises the human user and each user interacts with his or her BCS. We model a user's behaviour as an atomic component, as shown in Figure 5.

A user may also subscribe to one or more features. Each feature is modelled separately as an atomic component. Our telephony case study includes five common telephony features, taken from specifications for the Feature Interaction Contest [5]:

Call Forwarding (CF): The subcriber may specify a forwarding number to which all calls to the subscriber will be redirected.

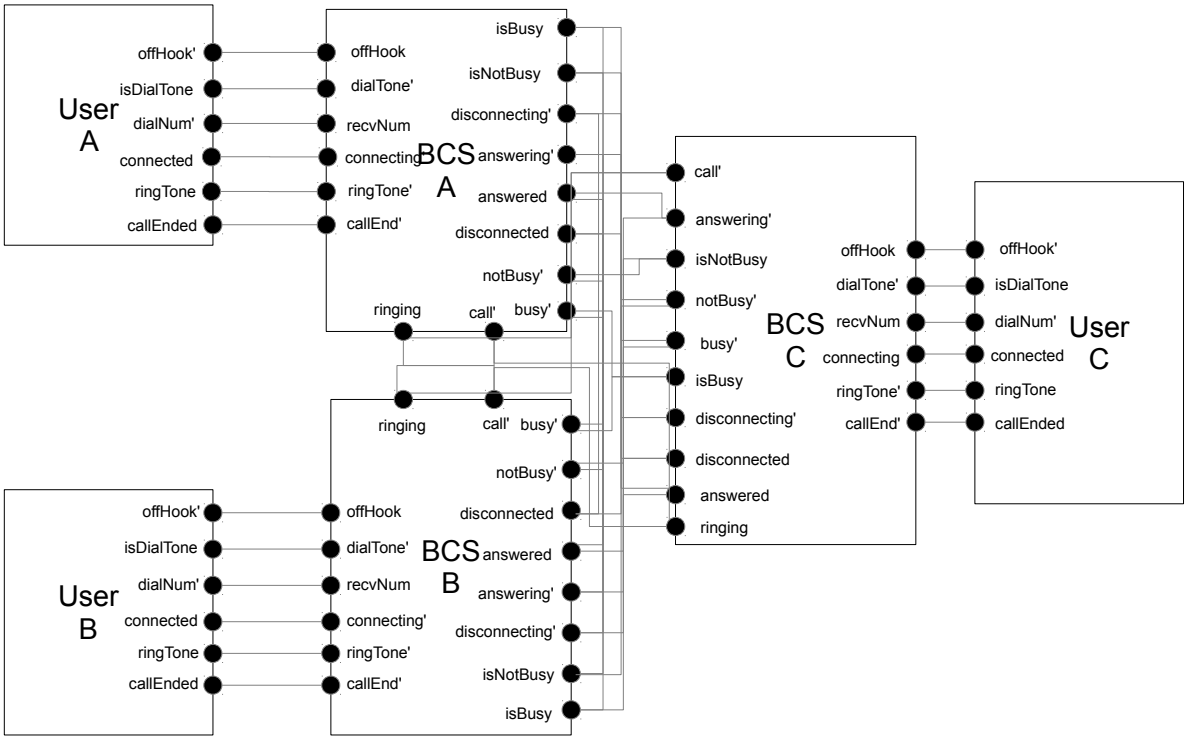


Fig. 4: Complete interaction-layer model of the basic-call service and user components for three users.

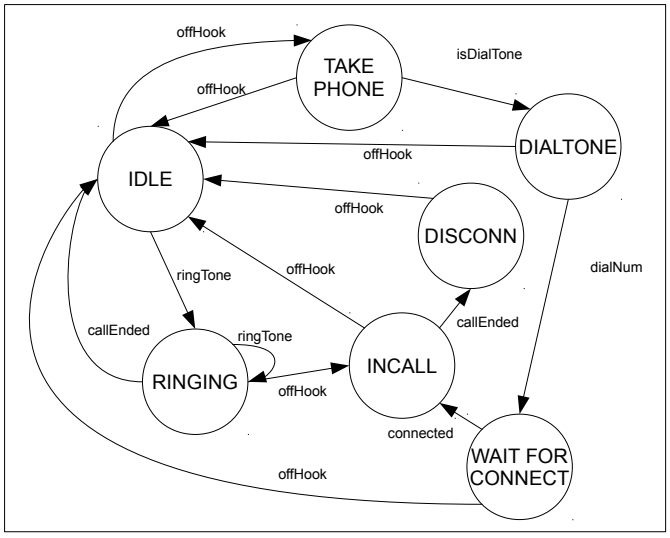


Fig. 5: The user environment, modelled as a BIP2 component.

Call Forwarding on Busy (CFB): If the subscriber receives a call when she is involved in another call, the feature will redirect the new call to a predetermined forwarding number.

Call Waiting (CW): If the subscriber receives a call when she is involved in another call, she may choose to put the original call on hold, answer the new call, and then toggle between the two calls.

Terminating Call Screening (TCS): This feature allows its subscriber to specify a list of blocked numbers. Any call originating from a number on this list will be terminated automatically.

Three-Way Calling (TWC): This feature allows a subscriber to add a third user to an existing call. Once three-way communication has been established, any user may chose to leave the call, resulting in a traditional two-way call configuration.

The BIP2 framework claims to support component-based modelling with an emphasis on inter-component interactions. The primary goal of our case study was to assess these claims in the context of feature-oriented modelling and feature interactions (FIs).

We evaluated BIP2's suitability for feature-oriented systems on three main points:

1) The overall complexity of a complete model of the telephony system (i.e., BCS together with the user model and optional features for each user) in terms of the number of states, components, transitions, connectors, interactions, and priorities.
2) The amount of work, in terms of new and changed BIP2 components, states, transitions, connectors, interactions, and priorities, that a developer must perform to add a new feature to an existing system. We also look at the difficulty of design decisions when composing new features in terms of limitations on the number or type of ports in existing components, transitions within the BCS component, and the types of existing connectors. We also strive to adhere to the principles of feature-oriented development. That is, the addition of a new feature to

the system should not require the modification of the BCS or existing features. Adherence to this principle allows for the independent and parallel development of features.

3) The difficulty of detecting and resolving FIs in terms of which ports, connectors, or transitions the modeller needs to consider to discover conflicting features and the number of changes they must make in the model to resolve these interactions.

Our secondary goal was to identify design methodologies or patterns for modelling and connecting BIP2 components in feature-oriented systems. In the next section we present three different feature-oriented modelling strategies and evaluate each of them based on the criteria above.

IV. Composition Strategies

Over the course of our case study, we identified and evaluated three distinct approaches to feature-oriented modelling in BIP2. In this section we describe these strategies, the thought processes that guided them, and our evaluations of their ability to combat the Feature Interaction Problem. See Appendix A for the full set of models we constructed throughout the course of this study.

Each approach tackles the problem of interconnecting features and the base system in a different way, resulting in different interactions, different degrees of model complexity, and different degrees of difficulty in the type of decisions the modeller makes during composition. We give a summary of our evaluations of each of these three approaches in Table I.

A. Reuse Approach

Our first approach to modelling and composing features stems from the idea that a feature overrides functionalities provided by the base service. In fact, our case-study features can naturally be described in terms of how they override functionalities in the BCS. CFB, CW, and TWC override the busy signal that the subscriber's BCS sends when she is currently occupied with another call by instead frowarding the incoming call, placing it on hold, or adding the incoming caller to the current call, respectively. CF and TCS both override how the BCS responds to an incoming call. When a user places a call to the subscriber, CF forwards the call automatically while TCS allows an incoming call to proceed only if the source number has not been blocked by the subscriber.

In BIP, the signals that these features override correspond to ports and interactions between BCSs. The busy signal is sent from a currently occupied user A to an incoming caller, user B, when the $busy'$ and $isBusy$ ports of their BCSs are enabled, respectively. Likewise, when user A places a call to an unoccupied user B, the $call'$ and $ringing$ ports of their respective BCSs are enabled, causing an interaction that triggers a ring tone to user B. To override the above functionalities, a feature must define a new connector with interactions that will execute in place of the existing interactions.

The reuse approach takes advantage of the BCSs' existing ports and connectors. Specifically, we integrate a new feature by first identifying the set of signals/behaviours it overrides, identifying the corresponding ports that send those signals, and then designing a new atomic component and interactions that either trigger or synchronize with these ports. We now give an example of how to add the CW feature to an existing system in the reuse approach.

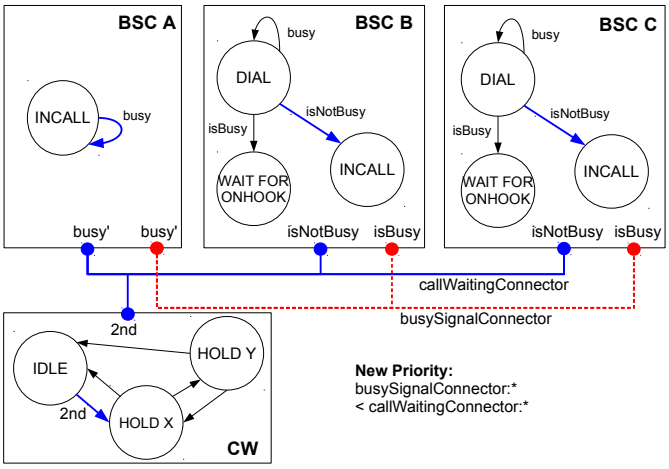


Fig. 6: The integration of CW in the reuse approach, components simplified for brevity. The original connector is shown in red and dashed, and the new connector, containing the $2nd$ port from the CW component is shown in blue and solid.

Example 1. CW overrides the busy signal sent by the subscriber's BCS. It allows the subscriber to place an incoming caller on hold and toggle between the new and existing caller, extending the behaviour of the INCALL state. When user A is busy, we want CW to allow a call from user C to progress to the INCALL state, and for the CW feature to indicate that user C is currently on hold, while user B is connected to user A. To accomplish this, we define a new connector that connects the $busy'$ port of user A to the $isNotBusy$ ports of users B and C and the $newcall$ port of the CW component. This connector overrides the existing connector (shown in red) that sends a busy signal from the BCS of user A to the call services of two other users, B and C.

If a new connector is defined during the integration of a feature, a priority must be put in place to give this connector precedence over the existing functionality. In our example, interactions in the new connector have higher priority and therefore execute in the place of interactions in the original connector.

Note that introducing a new connector often involves defining a new connector type in the BIP2 model that includes an extra port from the feature atom. When we add more features that override the same existing functionality, a single connector may contain ports from all BCSs and several feature components. The design of this new connector type must consider all possible combinations of enabled synchronizing ports; the number of interactions in this connector is then exponential in the number of features and BCSs that it connects. The

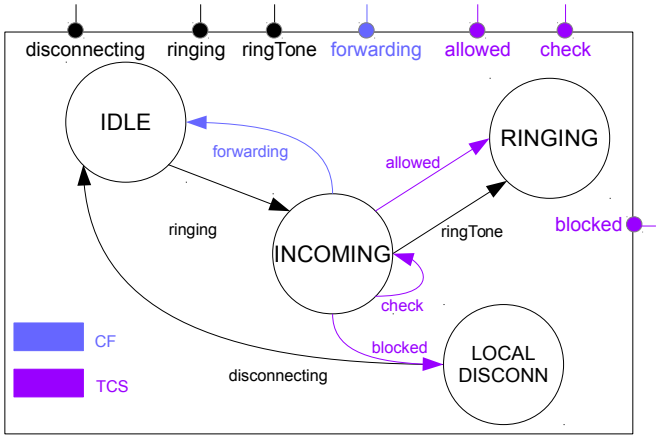


Fig. 7: A BCS component is rewired to support integration with TCS and CF. New transitions and ports for interacting with CF and TCS are shown in blue and purple, respectively.

modeller may have to introduce further priorities within the connector to resolve any undesired FIs.

We found that while the fully composed model may contain connectors with ports from several features, the modeller may still design these features independently. In the event that the new connectors of two or more features share the same ports from the BCSs of the subscriber and other users, the modeller may replace these connectors with a new connector that includes the BCS ports and the ports from all participating features. At this point, the modeller may resolve any resulting FIs with the introduction of additional priorities.

B. Rewire Approach

The reuse approach relies on existing ports and transitions in the BCS and prerequisite features to integrate new functionalities. As a result, the design of a new feature is limited by the structure of existing components. Furthermore, the reuse of ports results in connectors with a large number of ports and interactions, complicating the detection and resolution of FIs. The rewire approach explores a different method for integrating new features and gives the modeller a greater degree of flexibility by permitting modifications to existing components. In this approach, we relax our goal of adhering to the principles of object-oriented design to evaluate whether modelling freedom results in a reduction in the complexity of feature design and the detection and resolution of FIs.

In the rewire approach, we add a new feature to an existing system by first identifying the signals the feature sends or receives from the BCSs of the subscriber and connected users. For each signal, we add a corresponding port to the BCS component. Second, we decide the changes to the internal state of the BCS upon sending or receiving these signals, and add new transitions or places, accordingly. Finally, we design the feature component to trigger and synchronize with transitions in the BCS and connect it to the BCSs of the subscriber and connected users. The following example illustrates the process of adding TCS to an existing system.

Example 2. TCS screens all incoming calls against a list of blocked numbers. In order to do so, the TCS component communicates with existing components through the following signals:

- The BCS of the subscriber alerts TCS of an incoming call.
- TCS advises the subscriber's BCS to allow a call.
- TCS advises the subscriber's BCS to block a call.

We define three ports in the subscriber's BCS to send and receive the signals mentioned above: $check$, $allowed$, and $blocked$, respectively. Figure 7 shows the modifications to the BCS component. The original transitions are shown in black and the new transitions for TCS are given in purple.

Finally, we need to add priorities to ensure that the transitions of the new feature will execute. As stated in Section II, every place with more than one outgoing transition requires a priority ordering on these transitions. In this example, we introduce the priorities

$$check > blocked > allowed > ringTone$$

to ensure that a number is first checked, and then allowed to proceed if and only if it has not been blocked.

Note that we are not replacing or modifying connectors in this approach. As a result, each of the connectors we introduce will only connect the feature component and the existing BCS components, reducing the size of the connectors and the number of interactions the modeller needs to specify and consider when defining priorities. In the reuse approach, the maximum number of ports in a connector is linear in the number of features, resulting in an exponential number of possible interactions.

In addition to being small, the connectors in the rewire approach are also very similar in structure. Every connector that links a feature component with existing BCSs belongs to one of two connector types:

1) A connector with two ports: $trig'$ and $sync$. One port belongs to the feature component and the other belongs to the BCS of the subscriber or caller. This connector has two interactions: one where only $trig'$ is enabled, and one where both $trig'$ and $sync$ are enabled. The former should never execute and is given the guard $G_a : false$. The latter is given the guard $G_a : true$.
2) A connector with one triggering port, $trig$, and n synchronizing ports, $\{sync_1, \ldots, sync_n\}$, where n is the number of BCSs that links all BSCs and the new feature component. There are 2^n interactions in this connector, but only n are allowed to execute with the guard $G_a : trig.targetNum == sync_i.num$.

The BIP2 language allows the specification of parameterized connector types. When modelling our telephony system in this approach, we only need to define the interactions for the two generic connector types above. These connector types are later instantiated with specific port names, reducing the amount of work that goes into the specification of a BIP2 model.

Unfortunately, the advantages of the rewire approach come at the cost of violating the principles of feature-oriented development. The new feature components interact with BCS components through new ports. This means that the BCSs need to be extended with transitions that react to events on these ports. Thus, the addition of new features causes changes to the Petri nets of existing components. A feature's behaviour is no longer encapsulated in the feature component and new connectors.

C. Pipe-and-Filter Approach

The reuse and rewire approaches exemplify the challenge of feature-oriented modelling in BIP. There is a trade-off between modelling freedom and modularity; by treating components as black boxes, we restrict the ways in which other components can interact with them. In an effort to bridge the gap between these two strategies, we adapted an approach that standardizes how feature components interact with other features and the BCS. Interactions in BIP2 can be thought of as messages between components. Each component specifies the messages that it can receive (via synchronizing ports), and the messages it can send (via triggering ports). These messages may optionally carry data, such as the number of the synchronizing component and the target number of the triggering component. However, if we instead standardize the types of messages that features are allowed to send and receive, we make it much easier to interconnect features without knowledge of their internal structure. In the pipe-and-filter approach, components rely on the data passed through the ports of connectors to determine changes in their internal state, rather than on the names of ports and connected features.

We took inspiration for our third approach from the Distributed Feature Composition (DFC) architecture developed by Zave and Jackson [6] for the development and composition of telephony features. In DFC, feature components are connected sequentially between the BCSs of subscribing users. In DFC, information is piped through sequences of features as calls are placed from one BCS to another. The execution of features is serialized, with each feature triggering the next feature in the pipeline. A feature or service initiates or terminates a call by sending a *setup* or *teardown* message, respectively. Other messages such as *quickbusy* and *reserve* indicate that the target user is involved in a call and that the attempted connection has failed or succeeded, respectively. DFC provides a default resolution of FIs by imposing a priority ordering on the execution of features that determines which features will react first to messages passed through the pipeline and which features are in a position to specify the system's final response to a user request.

We made major changes to the original BSC to standardize the messages that are sent among components. We analyzed the communications between call services and grouped them into three main types of messages: messages that establish a call, busy messages that indicate the other service is currently unavailable, and disconnect messages that indicate one of the participants wishes to terminate the connection. We standardized messages and ports by encoding the type of message in the data that is exported through each port. Every interaction between an outgoing and incoming port passes the following data: (1) The enumerated message type (CONN, BUSY, or TERM), (2) the source number of the message, and (3) the target number of the message.

The original BSC components communicate and synchronize with each other through a number of different connectors, each representing a different step in the process of setting up, maintaining, and tearing down a call. The original BCS has a unique port for each kind of message (to setup and tear down connections, issue busy signals, or accept incoming calls). For each message, there are one or more transitions between places that keep track of the current state of he call. In the pipe-and-filter approach, each BCS and feature component has exactly two ports labelled in and out. These ports import and export data that specify the routing information of the call and the message type.

Example 3. Figure 8 shows the integration of TCS in the pipe-and-filter approach. The TCS component is placed between the BCS of the subscriber and the caller. In this position, it responds first to incoming calls. The caller attempts to establish a connection by triggering a transition in the TCS component and exporting the src, $dest$, and $code$ variables to indicate where the message is coming from, that the message is intended for the subscriber, and that the caller wishes to establish a voice connection. TCS will relay this message to its subscriber if and only if the caller, as indicated by src, has not been blocked. In the event that the call is blocked, TCS will instead send a message back to the caller indicating that the call has been terminated.

The standardization of messages allows features and call services to be oblivious to other components, while still reacting predictably to received information. Features can be designed independently and in parallel. This provides a greater degree of modularity than the rewire approach, which requires modifications to existing components, as well as the reuse approach, which requires modifications to existing connectors and interactions. Additionally, every feature has the same ports and is linked to other components with the same connectors, allowing us to make heavy use of parameterized connectors and further reducing the work of the modeller.

D. Evaluation of Approaches

We collected quantitative data on the complexity of our three approaches and give the results in Table I. We measured the work of the modeller to express and compose the features in terms of the number of additional places, transitions, data variables in the feature atoms, the number and complexity of the connectors needed to compose features, and the number of priorities introduced to mitigate FIs. We also measured the amount of rework to existing transitions, interactions, and priorities needed to properly compose a full model and resolve any unwanted FIs.

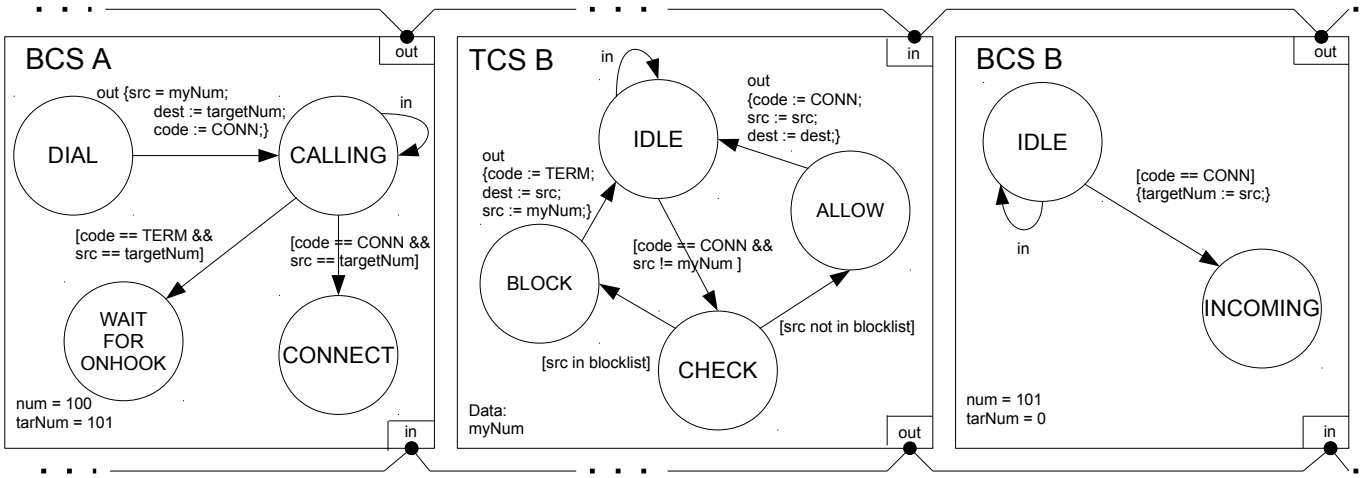


Fig. 8: A partial model of two BCSs and TCS connected in the pipe-and-filter approach.Components are connected sequentially, with interactions that carry connection and tear-down messages as data. Each feature in the chain has the ability to modify the messages that pass through it, changing the progression of the call.

TABLE I: Comparison of the overall complexity of a fully-composed BIP2 model in each of the three approaches. A fully-composed model has three users, each with a basic-call service, where one user has subscribed to all five optional features.

	Original BCS	Reuse approach	Rewire approach	Pipe-and-filter approach
feature places	0	13	15	**22**
feature transitions	0	16	18	**41**
BCS transitions	39	39	**75**	43
BCS data variables	5	5	5	**8**
connector types	2	**6**	2	3
defined interactions	6	**66**	6	6
connectors	16	22	**33**	16
priorities	19	26	**48**	0
reworked transitions	0	0	**1**	0
reworked interactions	0	**4**	0	0
reworked priorities	0	**4**	**4**	0

We evaluated each strategy on three main points: the complexity of the overall model (in terms of feature places and transitions as well modifications to the BCS and the connectors used to compose the overall model), the work of integrating a new feature into the existing model (in terms of additional feature components, connectors, and design decisions that require knowledge of existing components), and the difficulty of detecting and resolving feature interactions (in terms of analyzing existing components and the rework required to remove undesired behaviour).

Composed model complexity

As shown in Table I, each approach exhibits a different type of complexity. The reuse approach performs the most poorly in terms of the number of connector types and the number of defined interactions. The reuse of connectors requires the specification of an extreme number of interactions due to the number of ports and the many varieties of connector types. By contrast, both the rewire and the pipe-and-filter approaches use smaller and more similar connectors, resulting in the specification of fewer connector types and fewer defined interactions.

In the rewire approach, the simplification of connector definitions comes at the cost of the number of instantiated connectors, which requires the specification of more priorities, and may prove to be problematic for the model-checking or simulation tools available for BIP2 models. The pipe-and-filter approach performs the best in terms of connector complexity, with a similar number of defined interactions to the rewire approach and fewer priorities than both of the previous approaches.

In the reuse approach, the specification of a new feature component is restricted by the ports of existing features. Although this did not prove to be a significant problem in our case study, there may be more complex features that require a transition between places in the base service that are far apart. We predicted that the modelling freedom that resulted from allowing changes to existing components would ease the task of feature specification. However, our case study showed that the complexity of each feature component in terms of feature places and transitions does not vary significantly from the complexity of feature components in the reuse approach. The pipe-and-filter approach shows a small increase in model complexity in terms of the number of data variables in each component. Note that in previous two approaches,

a component receives only the messages it is interested in reacting to. In contrast, every feature in the pipe-and-filter approach receives all messages sent and received by the subscriber's BCS. While only a subset of these messages will be of interest to a particular feature, each feature must propagate any message it does not override to the next feature in the pipeline. This ensures that messages will reach the BSC endpoints unless they are explicitly intercepted and overridden by the intervening features, but increases the complexity of the feature components.

In the rewire approach, the monolithic implementation of features in the BCS increases the chance of introducing errors and complicates the task of debugging. We almost doubled the number of BCS transitions in adding only our five case study features. The advantages gained by simplifying the specification of connectors in the composed model are completely undone by the complexities introduced from revising the BCS with each new feature addition and resolving conflicts in the resulting monolithic component. We designed a completely new BCS model for the pipe-and-filter approach to accommodate standardized messages. However, the new BCS component is similar in complexity to the original BCS, with only slightly more transitions and data variables.

Integrating a new feature

The integration of a new feature in each of the three approaches requires varying amounts of knowledge and work. In the reuse approach, the modeller requires knowledge of, but does not make changes to, the components of the BCS and existing features. The places, transitions, and variables of the new feature component depend on two factors: (1) the transitions the feature needs to trigger in the BCSs of the subscriber and other connected users, and (2) the new states the feature needs to reflect. In the rewire approach, the modeller has the ability to add places and transitions to existing components. While the modeller has complete freedom over the structure of the feature component, all features require changes to the existing model and complex design decisions on how to make these changes without negatively impacting the overall desired behaviour of the system. Changes to the BCS, in the form of new transitions and priorities, must reflect desired behaviour, regardless of the combination of features present. Although the essential behaviour of each feature is modularized, the specification of features in this approach is spread out across multiple possibly unrelated components.

The standardization of messages in the pipe-and-filter approach allows features to be developed independently from existing components, relying only on the data variables passed through interactions to determine how the feature will respond to incoming or outgoing calls. This reduces the work of adding a new feature to specifying the feature's atomic component given our assumptions on the structure of these messages. The behaviour of a feature is determined by its reaction to the different types of messages sent and received by the BCSs. For example, CFB, CW, and TWC all react to a message of type BUSY issued by the subscriber's BCS. Each feature overrides its BCS by effectively intercepting the BCS busy signal and replacing it with its own signal: CFB issues a call to a new target and CW issues a connect message. A feature does not need to know which other features it is connected to; each feature only knows that its subscriber's BCS is upstream in the pipeline, and that the BCSs of other users are downstream. A feature component's design only depends on the type of messages it can receive from either direction. After a feature component has been designed, it must be added to the system. Since interactions between components are all the same, we make heavy use of BIP2 parameterized connector types.

Resolution of FIs

In the reuse approach, a FI occurs when the connector(s) of two or more features share the same port(s) in an existing component, or a connector contains ports from multiple features. To resolve the former, the modeller must decide a complete ordering on the conflicting features and introduce priorities to allow the interactions of higher priority features to take precedence over the interactions of lower priority features. To resolve the latter, the modeller must decide which interactions in the connector produce the desired behaviour, and prohibit all other interactions by setting the guard to false.

FIs in the rewire approach occur when transitions that leave the same source place in a component belong to different features. We partially resolve these interactions by adding priorities when we integrate the feature into the system. For example, CF and TCS conflict when the BCS is in the INCOMING place, as shown in figure 7. Our desired behaviour is to first screen the call against a list of blocked numbers before forwarding it. However, the resolution is not as simple as giving a complete priority ordering among all transitions from the INCOMING place. We wish both features to execute, not one in the place of the other. For this to work, we need TCS to perform the check, and then execute the $allowed$ transition in the BCS and return to the IDLE state before CF executes and forwards the call. To accomplish this, we need to move the CF $forwarding$ transition to the destination state of the TCS $allowed$ transition, requiring the rework of the modeller to resolve the FI.

Note that modifications to existing components are not necessary to resolve FIs in the reuse approach. We allow the modeller to redefine connectors to include two or more feature components. In this way, features can transition in the same execution step (e.g. TCS can return to the IDLE state and CF can forward the call in the same interaction). The resolution of features in the rewire approach is undesirable, as FI resolutions are realized as a monolithic solution in the BCS component.

FIs in the pipe-and-filter approach are resolved by feature placement in the pipeline between BCSs. The farther a feature is from the subscribing user's call service, the sooner it receives incoming messages and the more priority it has over other, closer features. A FI occurs when two or more features react to the same types of messages. For example,

both CF and TCS modify the connect signal sent to their subscriber's BCS. If TCS is farther downstream in the pipeline, it will be the first to receive a connect message and can check the source number against a list of blocked numbers before passing the connection request to the CF component. At this point, CF will forward the call to another target address. If instead TCS blocks the number, CF never receives the connection request. In this manner, the placement of features along the pipeline determines the order in which they react to signals, thereby using serialization to define a default resolution to FIs. The detection and resolution of FIs is reduced to identifying appropriate feature placement along the pipeline.

V. Discussion

As evidenced by the results shown in Table I, each approach exhibits complexity in a different aspect of the modelling process. The reuse approach adds features within the limits of existing components and ports, resulting in large, complex connectors. In the rewire approach, the modifications to existing components result in a large number of connectors and a complex, monolithic BCS that requires further modifications to resolve FIs. The pipe-and-filter approach standardizes component ports and messages to ease the task of feature integration, at the cost of more complex feature components.

It was a major challenge to support the development of features as black-box components. In BIP2, the specification of each component includes the specification of its ports, delineating the ways in which the component can interact with others. The specifications of ports, connectors, and interactions require some knowledge of the internal workings and ports of other components – which is the modus operandi in BIP2, where the modeller connects and specifies interactions among *known* components. In contrast, feature-oriented systems have a continuously evolving set of features. It is advantageous in this scenario for a feature developer to *not* know about the other features in the model. Called **feature obliviousness**, a feature necessarily knows about the base system that it is designed to extend or override, but ideally not about other features in the system. It is this obliviousness and separation of concerns that allows features to be developed in isolation and by third parties, and to be more easily integrated into an existing system without requiring significant rework of existing features or their connectors.

The reuse approach adheres most faithfully to the BIP2 vision for component-based design. Although the design of new features requires knowledge of existing components, we do not need to make changes to the existing components of the BCS and other features. Furthermore, while the large connectors in the reuse approach require an exponential number of specified interactions, FIs are easily detected and resolved by reworking existing connectors.

In contrast, the pipe-and-filter strategy is the most effective in supporting feature obliviousness. Not only does every feature have the same interface, but the connector types and their interactions are standardized. What is left to the modeller is to determine the order of connected features in the pipeline, and to instantiate the connectors to realize this pipeline. As a result, the composition of features and resolution of FIs was almost trivial. However, the feature components are more complex, because they need to extract information from the messages and because they need to process messages that they do not react to (i.e., they need to forward such messages to the next feature in the pipeline).

A. Threats to Validity

Although telephony systems provide a rich example of an extreme software product line, it is not representative of all feature-oriented systems. The comparative complexity of our three approaches may be different for other complex systems. In particular, we note that the pipe-and-filter strategy for connecting features works particularly well in telecommunications and Web services, where there is already a natural linear message path between communicating basic-call service components. Such a linear ordering of features may be less appropriate in other domains.

We chose the five features that constitute our case study based on their varying complexity and potential for interesting FIs. CW and TWC are among the more complex features in telephony, in terms of the number of places, transitions, and connectors. With respect to the features' prevalence to interact, three of the features override the BCS's busy signal, and the other two feature react to incoming calls. We believe this set of features is representative of end-user telephony features, but acknowledge that other classes of features (e.g. billing features) could produce different results.

VI. Related Work

Since the framing of the Feature Interaction Problem in 1989 [1], there have been myriad attempts to minimize the effort of the developer in composing systems that are prone to a large number of FIs [2], [7]. Off-line techniques aid the developer during the design and development of the system.

- Techniques for detecting FIs reduce the effort of the developer in discovering problematic compositions of features and pinpointing the sources of unexpected or undefined behaviour [8], [9], [10], [11].
- Filtering approaches limit the variability of a system by removing problematic or unlikely combinations of features from analysis, thereby reducing the number of FIs a developer needs to consider to those in a small set of feasible products [12], [13], [14].

On-line techniques for coordinating feature execution resolve FIs as they occur at runtime. Hay and Atlee proposed a specification and composition model that uses feature priority to automatically resolve FIs during composition [15]. Distributed Feature Composition (DFC) developed by Zave and Jackson [6] connects features into a network of pipe-and-filter architectures, avoiding FIs architecturally by serializing the features' executions. In contrast, BIP provides modellers

with the flexibility to specify how features are connected and prioritized.

There have been previous case studies that evaluate the modelling capabilities of BIP. Basu et al. performed a case study on wireless censor networks [16], the purpose of which was to assess whether the BIP2 framework is powerful enough to accurately model a distributed system composed of heterogeneous components. They also showed that the simulation and verification tools associated with BIP surpassed existing methods for observing the execution of the system and detecting errors. Bourgos et al. conducted a case study on the modelling of a MJPEG decoder [17]. They sought to accurately model and analyze the performance of embedded applications on different hardware platforms. The compositional nature of BIP allowed them to model the software and hardware architecture independently and later compose the components for a faithful representation of the mixed hardware/software system. Additionally, they found that the flexibility of BIP let them specify processor and bus policies, adding to the accuracy and detail of the overall system model. While these case studies provide evidence for the flexibility of BIP and its applicability to a wide variety of hardware and software systems, to our knowledge, there are no existing studies that analyze the use of BIP in the context of the Feature Interaction Problem. We provide both an analysis of its use to compose and analyze features and a comparison of design strategies for specifying feature-oriented systems in BIP.

VII. Conclusion

In summary, we investigated the effectiveness of BIP2 for modelling feature-rich systems, with particular attention to the amount of work needed to compose features, the amount of re-work needed to evolve a model to integrate new features, and the degree of complexity of the resulting model. Each of the three strategies that we studied has its advantages and its weaknesses, in terms of reducing the number of interactions that need to be written, or avoiding changes to the internal specifications of features, or simplifying the feature atomic components or the connector definitions.

Ultimately, when considering which strategies help to address the Feature Interaction Problem, the pipe-and-filter approach is the more effective design methodology: (1) it supports and preserves feature modularity, even when new features are added to the system, and (2) the amount of work and re-work needed to add a new feature is substantially less than in the other two strategies. An interesting side effect of this work is that we have shown how BIP2 – whose strength is in modular modelling and composition of components that are *designed to know about each other and to work together* – can be used to model components that *do **not know** about each other* and to compose them so that they can work together.

References

[1] T. Bowen, F. Dworack, C. Chow, N. Griffeth, G. Herman, and Y.-J. Lin, "The feature interaction problem in telecommunications systems," in *Proceedings of the 7th International Conference on Software Engineering for Telecommunication Switching Systems (SETSS)*, 1989, pp. 59–62.

[2] M. Calder, M. Kolberg, E. H. Magill, and S. Reiff-Marganiec, "Feature interaction: a critical review and considered forecast," *Computer Networks: The International Journal of Computer and Telecommunications Networking*, vol. 41, no. 1, pp. 115–141, 2003.

[3] A. Basu, S. Bensalem, M. Bozga, J. Combaz, M. Jaber, T.-H. Nguyen, and J. Sifakis, "Rigorous component-based system design using the bip framework," *IEEE Software*, vol. 28, no. 3, pp. 41–48, 2011.

[4] Verimag, "Bip2 documentation, release 2015.04 (rc7) [online]," Available: http://www-verimag.imag.fr/TOOLS/DCS/bip/doc/latest/pdf/BIP2.pdf (accessed November 5, 2015), Tech. Rep., 2015.

[5] M. Kolberg, E. H. Magill, D. Marples, and S. Reiff-Marganiec, "Second feature interaction contest," in *Proceedings of Feature Interaction Workship*, M. Calder and E. H. Magill, Eds. IOS Press, 2000, pp. 293–310.

[6] M. Jackson and P. Zave, "Distributed feature composition: a virtual architecture for telecommunications services," *IEEE Transactions on Software Engineering*, vol. 24, no. 10, pp. 831–847, October 1998.

[7] D. Keck and P. Kuehn, "The feature and service interaction problem in telecommunications systems: a survey," *IEEE Transactions on Software Engineering*, vol. 24, no. 10, pp. 779–796, October 1998.

[8] K. H. Braithwaite and J. M. Atlee, "Towards automated detection of feature interactions," in *Feature Interactions in Telecommunications Systems*. IOS Press, 1994, pp. 36–59.

[9] C. Prehofer, "An object-oriented approach to feature interaction," in *Feature Interactions in Telecommunications Systems IV*. IOS Press, 1997, pp. 313–325.

[10] B. Stepien, "Feature description and feature interaction analysis with use case maps and lotos," in *Feature Interactions in Telecommunications Systems VI*. IOS Press, 2000, pp. 274–289.

[11] S. Apel, A. von Rhein, T. Thüm, and C. Kästner, "Feature-interaction detection based on feature-based specifications," *Computer Networks*, vol. 57, no. 12, pp. 2399 – 2409, 2013.

[12] M. Heisel and J. Souquieres, "A heuristic approach to detect feature interactions in requirements," in *Feature Interactions in Telecommunications Systems V*. IOS Press, 1998, pp. 165–171.

[13] J. Bredereke, "Families of formal requirements in telephone switching," in *Feature Interactions in Telecommunications Systems VI*. IOS Press, 2000, pp. 257–273.

[14] D. O. Keck, "A tool for the identification of interaction-prone call scenarios," in *Feature Interactions in Telecommunications Systems V*. IOS Press, 1998, pp. 276–290.

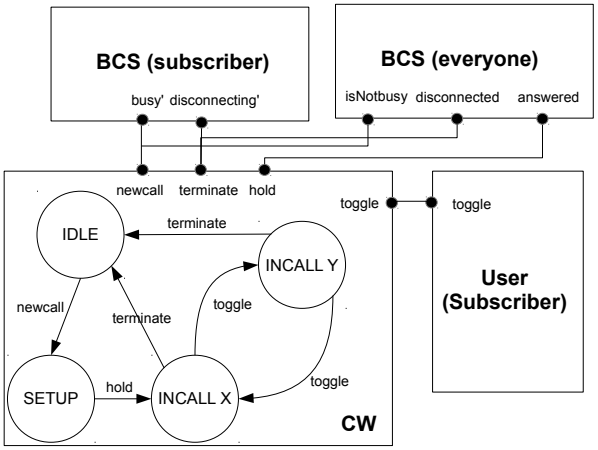
[15] J. Hay and J. Atlee, "Composing features and resolving interactions," in *Proceedings of ACM SIGSOFT Foundations of Software Engineering (FSE)*, 2000, pp. 110–119.

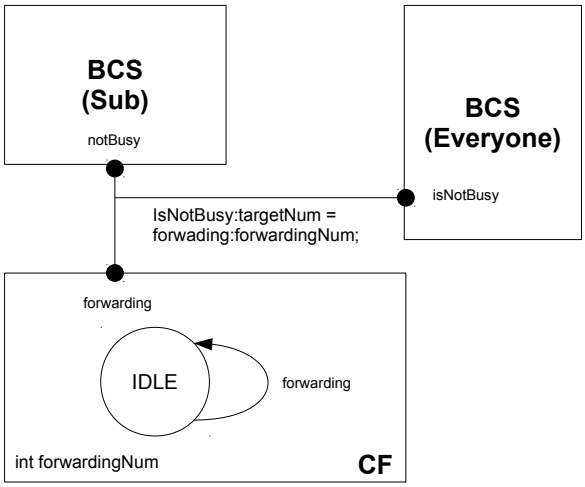
[16] A. Basu, L. Mounier, M. Poulhiès, J. Pulou, and J. Sifakis, "Using BIP for modeling and verification of networked systems - a case study on Tiny OS-based networks," in *Proceedings of the Sixth IEEE International Symposium on Network Computing and Applications*, Cambridge, USA, July 2007, pp. 257–260.

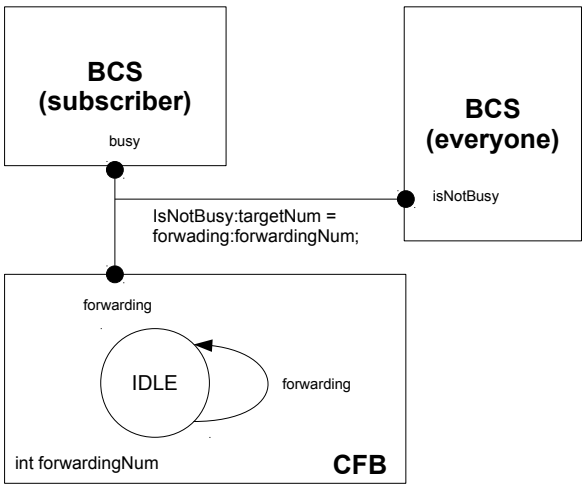
[17] P. Bourgos, A. Basu, S. Bensalem, K. Huang, and J. Sifakis, "Integrating architectural constraints in application software by source-to-source transformation in BIP," Verimag Research Report, Tech. Rep. TR-2011-1, January 2011.

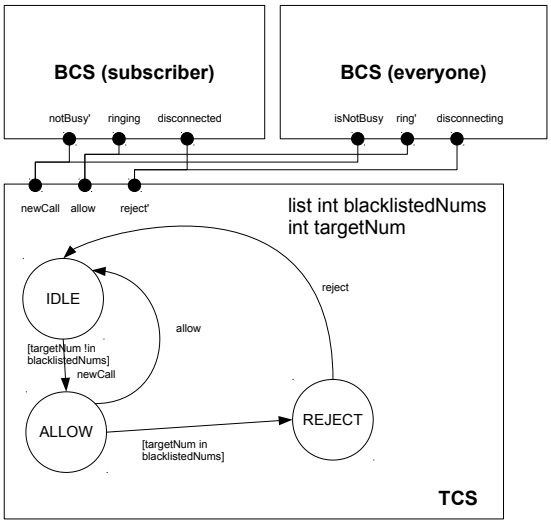
APPENDIX
CASE STUDY MODELS

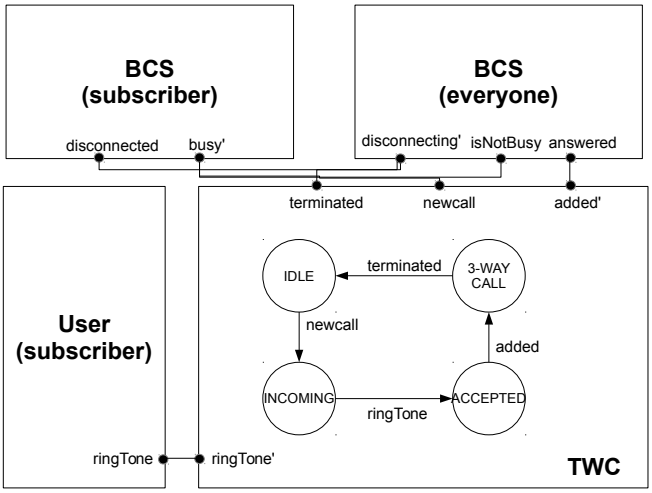
A. Models for the Reuse Approach





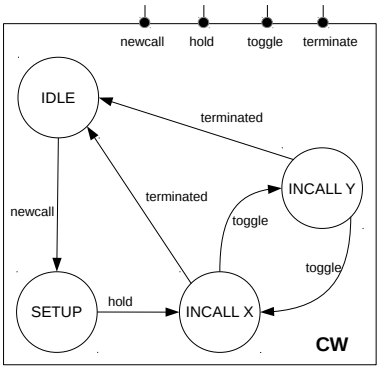


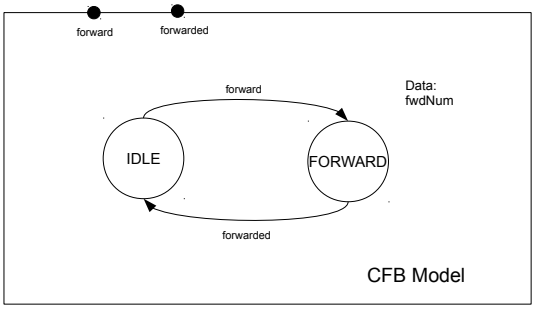


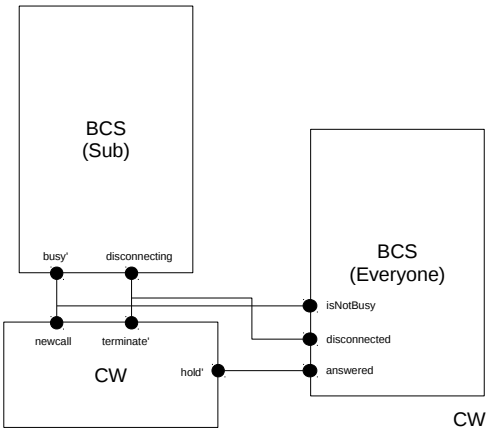


tcs_conn:* >
disconnecting_conn:*

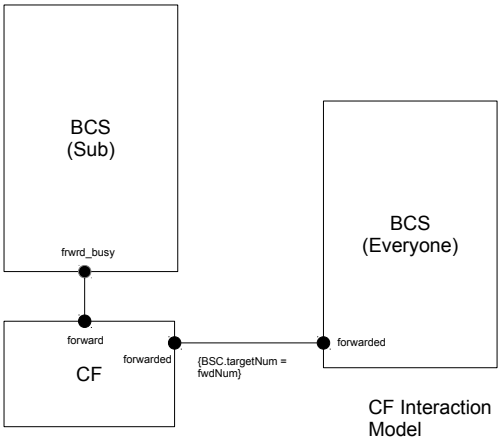
B. Models for the Rewire Approach



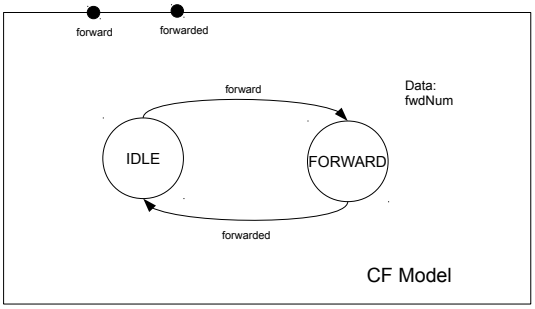


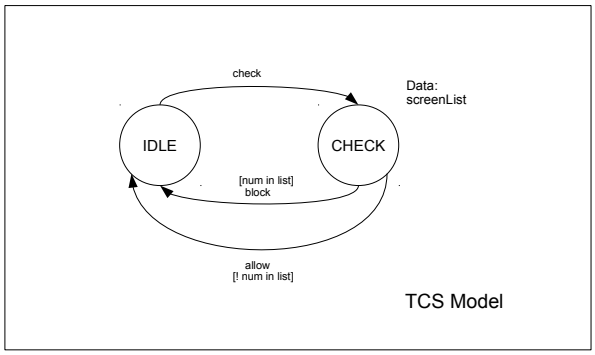


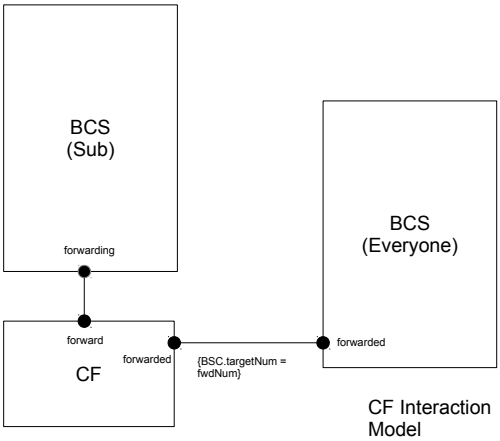
CW Interaction Model



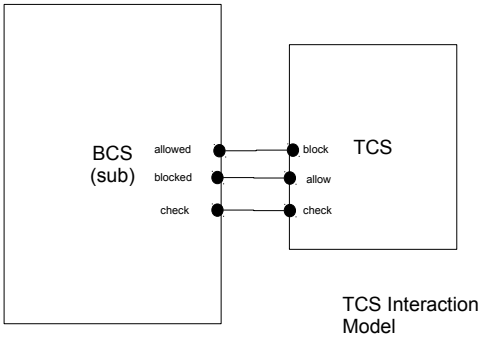
CF Interaction Model



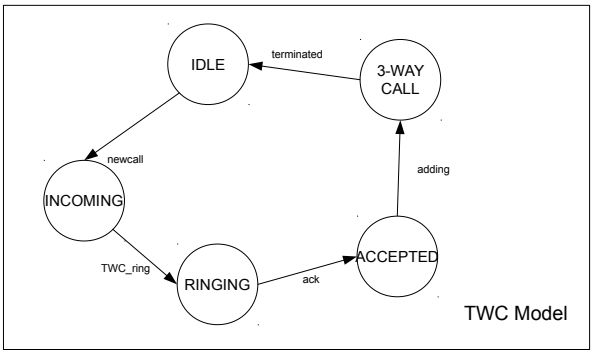


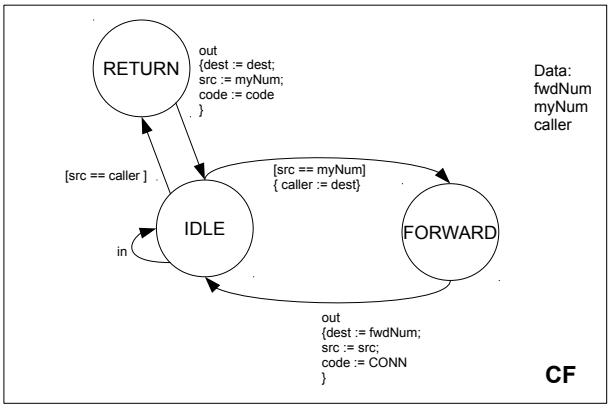


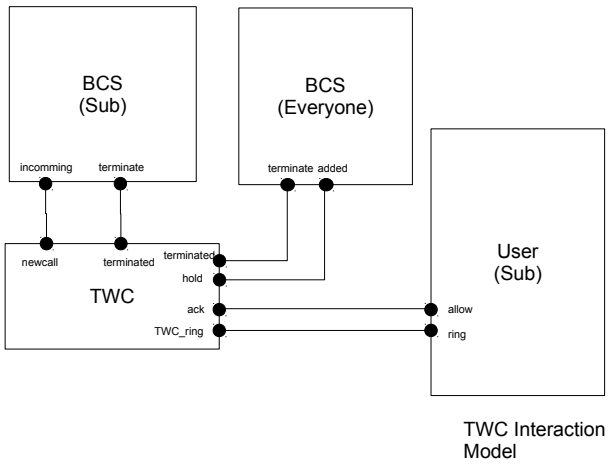
CF Interaction Model

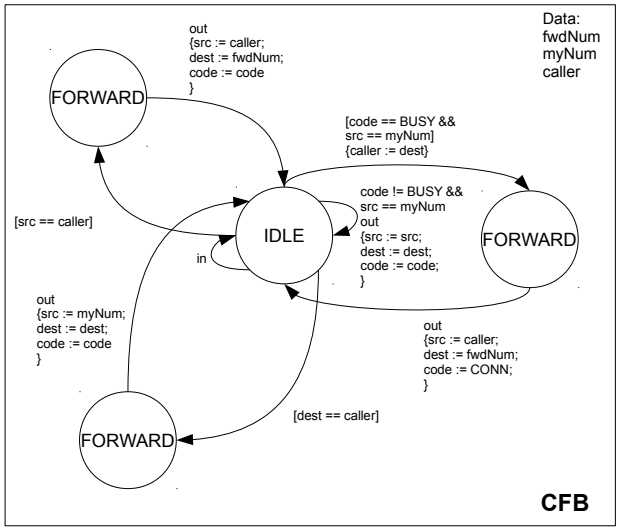


TCS Interaction Model

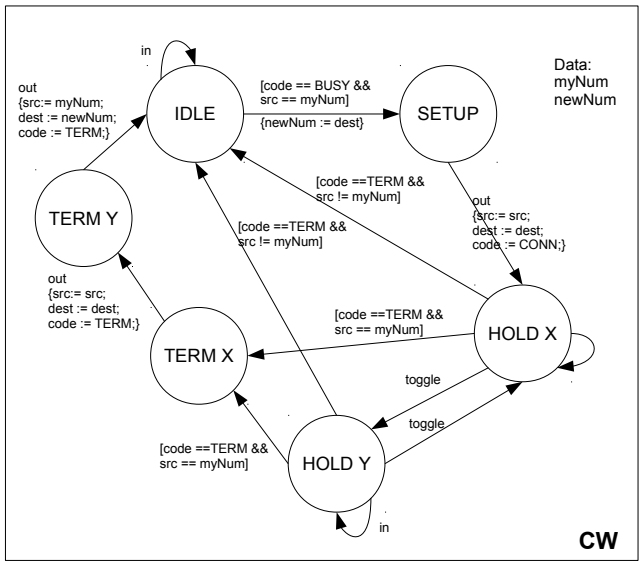


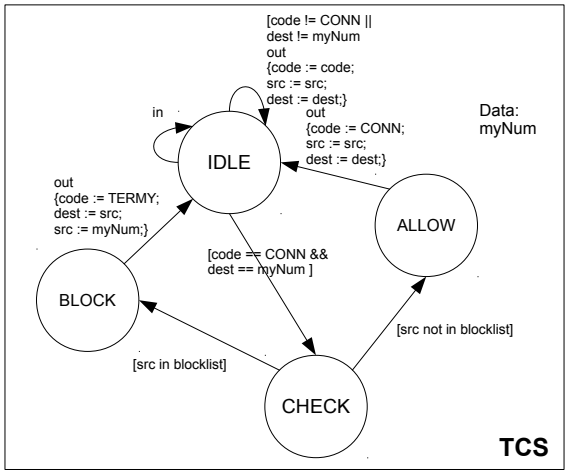


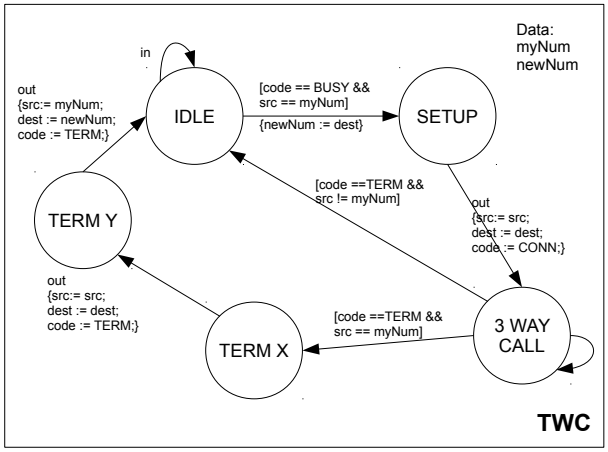




C. Models for the Pipe and Filter Approach






Data:
myNum
newNum
in
IDLE
out
{src:= myNum;
dest := newNum;
code := TERM;}
[code == BUSY &&
src == myNum]
{newNum := dest}
SETUP
[code ==TERM &&
src != myNum]
out
{src:= src;
dest := dest;
code := CONN;}
TERM Y
out
{src:= src;
dest := dest;
code := TERM;}
[code ==TERM &&
src == myNum]
3 WAY
CALL
TERM X
TWC