Compositional Reasoning for Port-based Distributed Systems

Alma L. Juarez Dominguez
School of Computer Science
University of Waterloo
Waterloo, ON, Canada
aljuarez@cs.uwaterloo.ca

Nancy A. Day
School of Computer Science
University of Waterloo
Waterloo, ON, Canada
nday@cs.uwaterloo.ca

ABSTRACT

Many distributed systems consist of components that run concurrently and communicate asynchronously through ports. The IP-based communication protocols used produce chains of connections in which the components communicate through queues. We present a compositional reasoning method to verify liveness properties of a communication protocol for chains of connections consisting of an unknown number of components. In our verification method, we model check components individually in an abstract environment, and then use the properties of the individual components in an inductive proof to conclude that the communication protocol properties hold for chains of connections. We describe how our method is used to verify properties of the call protocol used by the AT&T's Distributed Feature Composition (DFC) architecture.

Categories and Subject Descriptors

D.2.4 [**Software/Program Verification**]: Formal methods

1. INTRODUCTION

Many distributed systems consist of independent components that run concurrently and communicate asynchronously via first-in first-out queues. The protocols used often create graphs of connections between components where a component only communicates with its immediate neighbours. Parts of these graphs are linear chains of connections. Many IP-based telecommunications protocols are of this form. An example of such a system is AT&T's Distributed Feature Composition (DFC) architecture [11] for IP-based telecommunication services in which the components are telecommunication features such as call waiting and call forwarding. In this work, we are interested in proving liveness properties of a protocol for chains of connections of an unknown number of components representing by finite state processes that communicate asynchronously. Many of these properties take the form of ensuring that if a message is sent at one end of the chain, it will be received at the other end after passing through all the concurrent components of the chain. Automated verification of these types of queue-based systems is usually beyond the capacity of current verification tools, particularly when the number of components in the system is unknown. Thus, to reduce the verification complexity, we would like to be able to use compositional reasoning to verify these properties. Our contribution in this paper is in exploiting domain-specific information about these types of systems to produce a compositional reasoning method.

Much of the work on compositional reasoning techniques is applicable to a fixed number of heterogeneous components. Assume-guarantee reasoning [16] is designed to handle dependency problems between components. To prove properties of one component, we assume properties of another component and vice versa, which leads to proof rules in which care must be taken to avoid circular arguments. If the components are identical, induction in a theorem prover can be used to reason about an unknown, but bounded number of components. A combination of theorem proving and model checking has also been used to verify an unknown number of identical components (*e.g.,* [14]). Usually this work takes the form of using theorem proving to justify the reduction of the system to a fixed number of identical components.

We exploit domain-specific knowledge about both the properties to be proven and the system, and use a combination of theorem proving, model checking, and language containment to reduce considerably the verification effort. First, we recognize that while the components of the system are heterogeneous, to satisfy the protocol properties, they must all satisfy the same properties. We use the term *semiregular* to describe a set of components that behave the same with respect to the protocol but have distinct behaviours. This observation allows us to decompose the overall system properties into identical obligations that each individual component must satisfy. These individual properties are verified by model checking each component, and the properties are combined using induction in a theorem prover. The theorem proving effort is only in terms of properties of the components and properties of the queues, and does not need to be repeated as more components are added to the system.

The individual properties are unlikely to hold in every environment, but rather only when the component is placed in the environment of a chain of similar components us-

ing the same protocol. Here, we exploit a second aspect of domain-specific knowledge. The components in a chain communicate only with their immediate neighbours. A *port* is the communication behaviour that a component has with one neighbour sending and receiving messages. If we prove that a component works in an environment consisting only of the ports of its neighbours, then it will work in the entire chain. The port serves as an abstraction of not only its immediate neighbour but all components on that side of the chain. Furthermore, because of the semiregularity attribute, we are able find an abstract representation of a port to serve as the environment in which we check individual components rather than checking all combinations of a component with possible neighbouring ports. We use language containment to show that the behaviour of any port is contained within the behaviour of the abstract model of a port.

By exploiting these two domain-specific attributes (semiregularity and port-based communication), we have created a compositional reasoning method to verify liveness properties of a protocol used by chains of connections of an unknown number of components in a semiregular, port-based, asynchronous distributed system with bounded queues. An outline of the method is as follows: (1) use model checking to prove every component satisfies individual properties in an environment consisting of abstract ports; (2) use induction to prove the overall liveness property from the properties of the individual components; and (3) show that the behaviour of every port is contained within the behaviour of the abstract ports using a language containment proof. A key contribution here is isolating and abstracting the behaviour of a port of a neighbour, which both reduces the state space search for model checking, and means that we do not have to verify all the possible combinations of components that could be neighbours in a chain.

Handling asynchronous communication also offers challenges not usually addressed by assume-guarantee style reasoning which focuses on synchronous or broadcast communication. Queues increase the size of the state space. We tackle this problem by decomposing the verification of the properties of individual components into two parts. First, we use model checking to verify that the component satisfies the individual properties in a synchronous environment where the neighbouring components always accept output and send appropriate input. Synchronous communication abstracts away the queues and therefore reduces the verification complexity. Second, we verify that the component produces only output that is expected and receives all input provided in an environment consisting of abstract representations of the ports of its neighbours and asynchronous communication with a queue of bounded size. This check is basically a check for lack of deadlock and is reusable for multiple properties. In the theorem proving component of our method, we abstract the behaviour of the queues to perfect communication: every message that is sent is eventually ready to be received.

We demonstrate the utility of our approach by describing a significant case study showing how our method works to prove protocol properties of the Distributed Feature Composition (DFC) architecture. DFC is used for coordinating telecommunications features. In this system, components are features such as call waiting. Users can subscribe to features, which creates the situation where the number of features to be used in a call is unknown. DFC creates chains of components that run concurrently and communicate only with their neighbours through queues. An interesting aspect of this case study is that we created a hierarchy of abstract ports. This hierarchy allows feature behaviour to be compared to the most appropriate element in the hierarchy when checking language containment.

In the next section, we describe our compositional reasoning method. In Section 3, we describe our case study applying the method to DFC. We first provide an introduction to the DFC architecture, and present our model of DFC. We also include a brief discussion of how we model checked instances of DFC with a fixed number components to help debug our protocol properties. Then, we describe how we use our compositional reasoning method for DFC. We conclude with a discussion of related work, a summary and a brief description of future work.

2. COMPOSITIONAL REASONING

The goal of our work is to verify overall system properties of a communication protocol in all chains of connections with any number of components. We decompose the problem into two main parts: verify properties of individual components, which we will call *individual properties*, and then assuming all components satisfy the individual properties, prove the overall system property. Figure 1 illustrates our compositional reasoning method. Verifying components individually is done in steps (1), (2), and (3) using model checking and language containment. Proving the overall system property is done using induction over the structure of the chain of components in a theorem prover in step (4).

The proof obligations of steps (1), (2), and (3) allow us to conclude that a component will satisfy the individual properties in *any* chain of components of the system. These steps must be completed for every component. Because the system is semiregular and uses port-based communication, we create an abstract model of a port and use this abstraction to represent the behaviour of the neighbouring port plus all aspects of the chain on the other side of the neighbour. The abstract model of the port captures that most general behaviour of a port. In step (1), which we call *port compliance*, we verify that the behaviours of each port of every components is within the behaviour of this abstract port.

Unlike broadcast communication, port-based communication over chains of components means that a component will only communicate with its immediate neighbours. There could be multiple neighbours, but there is usually a reasonably small, fixed maximum number of neighbours for a component. For example, a call waiting feature communicates with at most four neighbours: a subscriber to the feature, the user to whom the subscriber is talking, a user on hold, and any other user who receives a busy signal. We decompose the step of verifying the protocol properties of a component into two parts to reduce the state space. First, we show *protocol compliance* in step (2), which means that the component satisfies the individual properties in a cooperative environment using synchronous communication. Second, we show *I/O compliance* in step (3), which means that the output produced by the component is expected by its environment and the environment provides the input expected by the component. This separation results in a reduced state

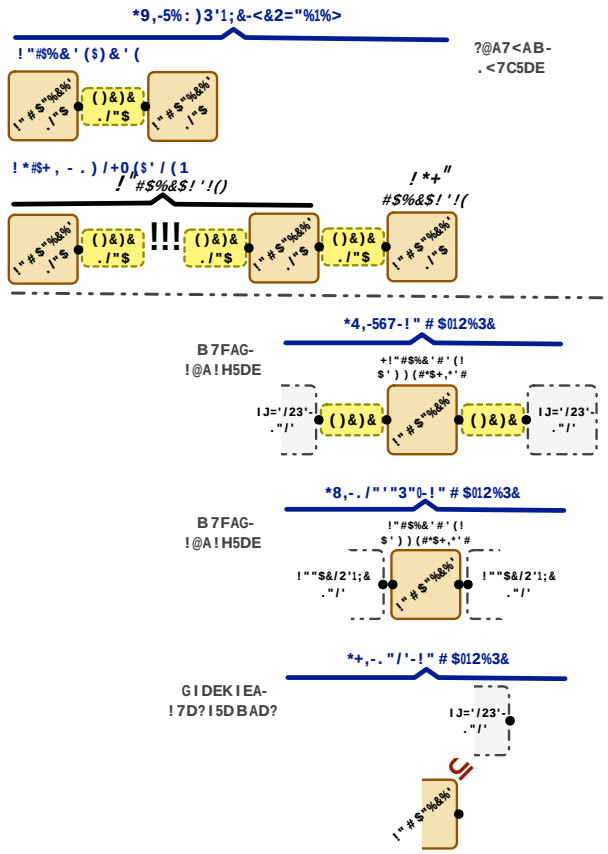

Figure 1: Compositional Reasoning Method

space for model checking. Also, we expect the I/O compliance step to be reusable for multiple individual properties (and therefore multiple overall system properties). Next, we provide further details on these steps.

In the *port compliance* step, we prove, using language containment, that the behaviour of each port in a component is within the behaviours of the abstract port. First, we isolate the behaviour of the component to its communication on only one port by replacing all transition triggers except those dealing with communication on this port with a guard of "true" and removing all outputs except those to the port being verified. This is a valid abstraction of the port's behaviour – it does not add or remove any port behaviour. Second, we find an abstraction function, abs, matching the states of the component (`concrete`) with the states of the abstract port (`abstract`). Then we show, for every transition in the concrete machine consisting of a source state (src), destination state (dest), and a trigger (sig), which involves receiving or sending a signal, that:

$$\forall\, \mathsf{src}, \mathsf{sig}, \mathsf{dest} \cdot (\mathsf{src}, \mathsf{sig}, \mathsf{dest}) \in \texttt{concrete} \Rightarrow (\mathsf{abs}(\mathsf{src}), \mathsf{sig}, \mathsf{abs}(\mathsf{dest})) \in \texttt{abstract}$$

In the *protocol compliance* step, we use model checking to verify the individual properties of a component using cooperative ports and synchronous communication. A *cooperative port* sends and receives any signal the component needs during its execution. It allows us to hide the behaviour of the environment by assuming the environment will cooperate. The use of synchronous communication abstracts away the behaviour of the unbounded queues, which considerably reduces the state space during the verification effort.

In the *I/O compliance* step, we use model checking to verify that a component communicating with neighbouring components asynchronously over a channel of bounded length receives only the signals it is expecting and sends only the signals expected by the environment. In this step, we use abstract ports as the neighbours of the component. We check for a combination of lack of deadlock (invalid end states) and termination with empty queues. No other components in the chain need to be considered because the component only communicates with its immediate neighbours.

The size of queues needed between the components for checking I/O compliance will depend on the system being verified. In our DFC case study, we required a queue size of only one to prove lack of deadlock and termination with empty queues. While a larger queue size could have been used, the smaller queue size reduces the state space and forces the maximum number of interleavings. Having a larger queue size would permit extra behaviours that involved fewer interleavings.

When carried out for every component, we can conclude from the above three steps that every component in the chain satisfies the same set of individual properties with any neighbours the system might provide to it in the chain.

In step (4), we use induction in a theorem prover to prove the overall system properties. The induction is on the structure of the chain of components. We assume the individual properties hold of all components and that the queues provide perfect communication: any message that is sent on a queue will eventually reach the component's neighbour. The base case is two components connected by a queue. In the inductive step, we assume the overall system property holds if there are n components in the chain, and then prove the overall system property will hold if there are $n+1$ components in the chain. The inductive reasoning is performed in terms of properties only, and is only performed once. Since we are not working with models of components in this step, there is no state space search.

Currently, we use the SPIN model checker [10] because it supports both synchronous and asynchronous communication and, therefore, we describe the properties in linear temporal logic (LTL). We use the HOL theorem prover [7], and a simple tool that we wrote for checking language containment. In our case study (described next), we determined the properties of individual boxes by hand, although this was quite straightforward given the overall system properties of interest. Also, we arrived at the abstract port manually through trial and error, but it is the union of the possible behaviours of the components' ports.

3. CASE STUDY: DFC

In this section, we describe how our method is applied to DFC. Our model of DFC is based on the material found in the DFC Manual [12], DFC modifications [26, 25], and various papers describing the architecture [11, 21, 27].

3.1 DFC Model

The Distributed Feature Composition (DFC) system is an architecture developed by Jackson and Zave at AT&T for co-

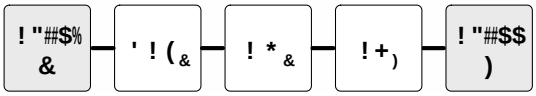

Figure 2: Example of a Usage

ordinating telecommunication features [11]. A *feature box* is a function for the users of a system that is performed on top of basic services. An example of a feature in the telephony domain is call waiting. The DFC architecture is a distributed system in which each feature runs independently. A DFC *usage* describes the response to a request for a telecommunication service at certain time. A usage can be viewed as a graph that consists of the features subscribed to by users assembled in an order based on precedence information. An example of a usage is presented in Figure 2. The nodes of the graph are features and the edges are bidirectional communication channels[1], which are unbounded first-in, first-out bidirectional queues between communicating boxes. Communication occurs only between features that are immediate neighbours. A feature, such as call waiting, can communicate with more than two neighbouring features and therefore create a branching usage.

A *box* is a process that performs either interface or feature functions. *Interface boxes* (*e.g.,* caller or callee) provide an interface to physical devices to communicate to users or to other networks. *Feature boxes* are either free or bound. A *free* feature box is one for which a new instance of it is generated every time the feature is to be included in a usage. An example of a free feature box is call forwarding, which is not persistent and gets created upon request. A *bound* feature box is dedicated to a particular address, and even if it is already in use within an existing usage, the same feature box is made part of a new usage. An example of a bound feature box is call waiting. If a call waiting box is involved in a usage, and the subscriber is called by another caller, the route for this second call goes through the call waiting box that is already in use.

We model each box (feature boxes and interface boxes) as a single process in PROMELA, the modelling language of SPIN. The processes communicate via channels. There are three phases to the interaction between caller and callee: setup, communication, and teardown. In the *setup phase*, each internal call is set up in a triangular and piecewise manner as illustrated in Figure 3. A `setup` signal[2] from a box first goes to the router on the `box_out` channel (step 1), then the router determines the next box in the usage and sends it a `setup` signal on the `box_in` channel (step 2), and finally, a communication channel (`ch1`) is created between the first two boxes in the usage (step 3). In step 2, the router determines the next box in the sequence based on the caller and callee's subscriptions, and precedence information ordering the boxes in a usage. When a box receives a `setup` signal from the router, it sends an `upack` signal to the calling box along the communication channel connecting the two boxes.

[1]These are called internal calls in the DFC literature.

[2]A `setup` signal contains additional information such as the source and destination. These details are not needed for the properties of interest in this work.

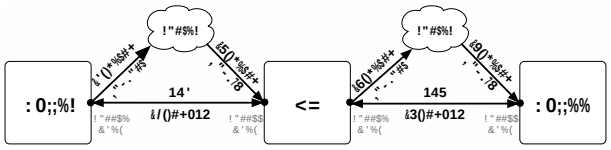

Figure 3: Setup Phase

The setup phase continues with the second box in the usage sending a `setup` signal to the router to be forwarded to the next box in the usage. The end of the communication channel connected to the box that initiates a call by sending a `setup` is called a *caller port*, and the port on the other end of the communication channel is called a *callee port*. Once a usage from a caller process to a callee process has been set up, the caller process can respond with an `avail` (available) or `unavail` (unavailable) signal to indicate its status. There are other status signals (`unknown` and `none`) that are part of the DFC call protocol, which are not implemented in our model. The call then proceeds to either its communication phase, or directly to its teardown phase.

In the *communication phase*, data is exchanged between the caller and callee. In the *teardown phase*, the usage is destroyed. Similar to the setup phase, the teardown phase is performed piecewise: a `teardown` is acknowledged by sending a `downack` back to the box that sent the `teardown`, and then propagating the `teardown` to the next box in the usage. In contrast to `setup` signals, `teardown` signals can come from either end.

Processes in PROMELA run in an interleaved manner. Statements in processes can be grouped using an `atomic` construct, which means these statements cannot be interrupted. Unlike previous models of DFC created in SPIN (e.g., [22]), in our model, instances of boxes in a usage are *dynamically* created by the router as they are needed. Using dynamic process creation matches the way DFC works more closely than having persistent processes for features. It also results in a significantly simpler model than having all processes persist and having channels for all possible connections. We have an array of channels that are allocated by the router as the usage is created. The router checks subscription information and runs a new instance of a free feature box process, or directs communication to a bound feature box or an interface box as appropriate. As the DFC routing protocol has been analyzed previously [20, 25], we do not model details of the router and instead dynamically create local instances of a router process as needed. We model the communication to and from the router in an atomic sequence using rendezvous channels for `box_out` and `box_in`. No other process can execute during this atomic sequence of actions (processes communicating through zero-capacity channels never block) and once completed, the router has reached the end of its code. Therefore, it is never necessary to have two router processes in existence at the same time, which means the same channels, `box_out` and `box_in`, can be used for all communication with the routers. This method reduces the state space of the model.

The communication that occurs on the communication channels connecting the caller and callee ports of a box may be delayed, *i.e.,* a signal may not be read immediately after it was sent because of the interleaved execution of processes. Therefore, these channels are not zero-capacity. DFC as-

sumes that signals are read in the same order as they were sent on a particular channel, and PROMELA's channels have this behaviour.

We have modelled four feature boxes: the free transparent feature (FTF), the free feature boxes call forwarding (CF) and originating call screening (OCS), and the bound feature box call waiting (CW). Our complete PROMELA model is available in [13]. Call waiting is the most complicated of the features we modelled, and its state transition diagram contains 338 states and 445 transitions. It has three ports on which it may communicate (two users talking and one on hold). There is also an additional port called `busy`, which rejects the connection when a fourth user tries to communicate.

3.2 Categorization of Boxes

Some DFC boxes have the power to change the topology of a chain of components by placing, receiving, or tearing down calls. To describe properties of DFC chains, we created the following characterization of boxes:

***User agents* (*UA*):** Set of interface boxes, plus the feature boxes that can act like a user. A user agent box can request the creation of a chain of components (*e.g.*, call forwarding on busy), or respond to such a request. The response can be positive, accepting the creation of a chain of components (by the generation of an `avail` signal), or negative, stopping the continuation of a chain (by the generation of an `unavail` signal). A user agent such as call waiting can create a branch in a chain.

***Transparent* (*T*):** [3] Set of all boxes that must forward any call protocol signals that they receive onto the next box in the usage. These boxes may not create new chains of components.

Of the features we modelled, call forwarding and the free transparent feature box are transparent boxes. Call waiting, originating call screening, the caller process, and the callee process are user agents.

We denote as a *segment* any part of a usage that is a chain of components which starts at a user agent box and ends at a user agent box. A new `setup` (rather than a propagated `setup` signal) is involved in the creation of a branch of a usage, therefore every `setup` signal generates a single segment. Segments can be connected together at user agents as illustrated in Figure 4.

3.3 Segment Properties

In this section, we describe our call protocol properties of segments, which stitch together behaviour of individual boxes to provide a concise statement of end-to-end behaviour of DFC, where the "ends" are user agents. The general form of the properties is "After a user agent *sends* a signal, the user agent at the other end of the segment eventually *receives* the signal." To state the segment properties, we use the names of the channels shown in Figure 5. We call the sender of a `setup` signal at one end of the segment an

[3]There are specific features called the free transparent feature (FTF) and the bound transparent feature (BTF) boxes. The category of "transparent" boxes includes more than just these particular feature boxes, although the category name is inspired by the behaviour of FTB and BTF.

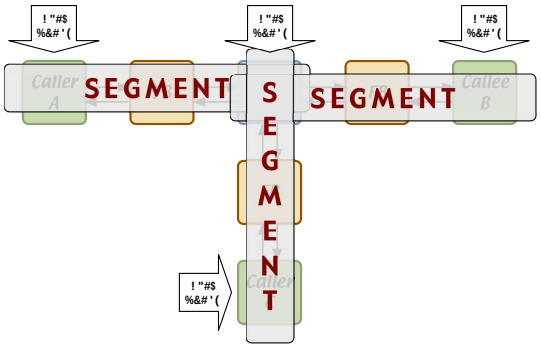

Figure 4: Usage composed of Segments

$\Box$(box_out!setup $\Rightarrow$ $\Diamond$box_in?setup)	***S*.1**
$\Box$(box_out!setup $\Rightarrow$ (($\Box\neg(\exists$i.(i.ch2)?teardown)) $\Rightarrow$ (alpha!teardown $\Rightarrow$ $\Diamond$(beta?teardown))))	***S*.2**
$\Box$(box_out!setup $\Rightarrow$ (($\Box\neg(\exists$i.(i.ch1)?teardown)) $\Rightarrow$ (beta!teardown $\Rightarrow$ $\Diamond$(alpha?teardown))))	***S*.3**
$\Box$(beta!avail $\Rightarrow$ $\Diamond$alpha?avail)	***S*.4**
$\Box$(beta!unavail $\Rightarrow$ $\Diamond$alpha?unavail)	***S*.5**

$\Box$ means "always"; $\bigcup$ means "strong until"
$\Diamond$ means "eventually"
`ch!sig` means signal `sig` is sent on channel `ch`
`ch?sig` means signal `sig` is received on channel `ch`
`i.ch` means channel `ch` of box `i`

Table 1: Segment Properties

Upstream User Agent (UUA), and the receiving user agent is called a *Downstream User Agent (DUA)*. We also need one feature box (FB i) in the middle of the segment to state the properties.

The segment properties, to be checked for segments with any number of feature boxes, are formalized in linear temporal logic (LTL) with predicate logic in Table 1. The first property (***S*.1**) concerns the propagation of a `setup` signal from the upstream user agent to the downstream user agent. The second and third properties (***S*.2**, ***S*.3**) describe the segment behaviour for a `teardown` signal. A `teardown` signal may originate from either end of the segment after a `setup` signal has been sent. A `teardown` signal propagates to the end of a segment only if a teardown has not been sent by the other end of the segment and received at some intermediate box. A DFC feature that receives a `teardown` from one side when it has already received a teardown from the other side does not propagate the signal. To express this property, we introduce an intermediate box in the segment. The final two properties (***S*.4**, ***S*.5**) describe the propagation of `avail` and `unavail` signals from a DUA to a UUA. By checking all the segments that compose a usage, we verify the behaviour of the usage.

3.4 Model Checking Fixed Configurations

We began our verification effort by model checking fixed configurations of DFC, *i.e.,* models with a particular number of callers and callees processes, and fixed subscription

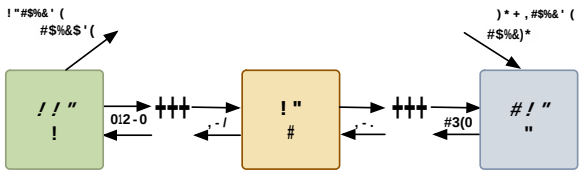

Figure 5: Channels used in Segment Properties

information.

By using dynamic process creation, we had a model of infinite size[4], thus we limited our callers to calling at most twice. Once call waiting was introduced into usages, the state space explosion became apparent and the largest model we could check was CW with three users and no other features. Each user could make one call. This model checking effort took 28 minutes on a 1.4 GHz Xeon CPU with 4GB of RAM and reached a maximum depth in Spin of 5 million steps. All the results that we report in this paper were produced on this equipment. Because Spin uses an explicit state representation, the depth is a measure of how many states have been explored. Any larger configurations that we tried to check exceeded the maximum number of steps possible within the memory available, and therefore did not complete. However, by checking fixed configurations, we could debug both our model and the segment properties. This debugging exercise was very useful as we discovered a previously unknown race condition in call waiting within five minutes while checking for deadlock. The new behaviour was added to the original specification of CW, and the final description is used in all our verifications.

3.5 Compositional Verification for DFC

In this section, we describe how our compositional method is used for verifying the properties of Table 1 on DFC segments of unknown, but finite length. In the following subsections, we provide details on each of the proof steps.

3.5.1 *Protocol Compliance*

First, we determined the individual properties that capture the essential box behaviour sufficient to prove the segment properties. The DFC Box Properties are listed in Table 2 and are described in terms of the channels shown in Figure 6. Most of these properties have the form "After *receiving* a signal, the box eventually *sends* a signal on another channel". Most box properties are similar to the segment properties (Table 1), but reflect what an individual box must do so that the segment properties will be satisfied. For example, Property ***T*.5** describes the behaviour that after the usage is setup, the first `teardown` received from either neighbour is forwarded. Property ***D*.3** requires a send of an `upack` signal to be followed by a send of either an `unavail` or an `avail` signal unless a `teardown` is received.

We check these properties using model checking in Spin.

[4]Even though there are only a fixed number of features, callers, and callees, Spin processes created dynamically all have unique process identifiers and there is always an interleaving order that allows a process to persist forever. Thus a usage that had previously been explored would be explored again because of different process identifiers.

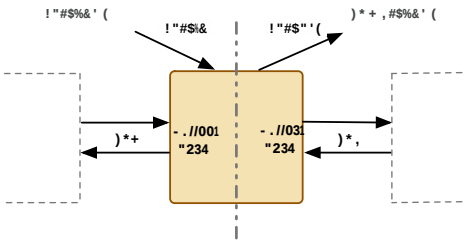

Figure 6: Channels and Ports for a Two-way Box

Transparent Box Properties	
$\Box$(`box_in?setup` $\Rightarrow$ $\Diamond$`ch1!upack`)	***T*.1**
$\Box$(`box_in?setup` $\Rightarrow$ $\Diamond$`box_out!setup`)	***T*.2**
$\Box$(`ch1?teardown` $\Rightarrow$ $\Diamond$`ch1!downack`)	***T*.3**
$\Box$(`ch2?teardown` $\Rightarrow$ $\Diamond$`ch2!downack`)	***T*.4**
$\Box$(`box_in?setup` $\Rightarrow$	
((¬`ch2?teardown` $\bigcup$	
(`ch1?teardown` $\wedge$ $\Diamond$`ch2!teardown`))	
$\vee$ (¬`ch1?teardown` $\bigcup$	
(`ch2?teardown` $\wedge$ $\Diamond$`ch1!teardown`))	
$\vee$ $\Box$(¬`ch1?teardown` $\wedge$ ¬`ch2?teardown`))	***T*.5**
$\Box$(`ch2?avail` $\Rightarrow$ $\Diamond$`ch1!avail`)	***T*.6**
$\Box$(`ch2?unavail` $\Rightarrow$ $\Diamond$`ch1!unavail`)	***T*.7**
Upstream User Agent Box Property	
$\Box$(`ch2?teardown` $\Rightarrow$ $\Diamond$`ch2!downack`)	***U*.1**
Downstream User Agent Box Properties	
$\Box$(`box_in?setup` $\Rightarrow$ $\Diamond$`ch1!upack`)	***D*.1**
$\Box$(`ch1?teardown` $\Rightarrow$ $\Diamond$`ch1!downack`)	***D*.2**
$\Box$(`ch1!upack` $\Rightarrow$	
((¬`ch1?teardown`) $\bigcup$	
(`ch1?teardown` $\vee$ `ch1!avail` $\vee$ `ch1!unavail`))	***D*.3**
$\Box$(`ch1!unavail` $\Rightarrow$	
((¬`ch1?teardown`) $\bigcup$	
(`ch1?teardown` $\vee$ `ch1!teardown`))	***D*.4**

Table 2: DFC Box Properties

We place each box in a cooperative port environment that can receive and send non-deterministically *any* of the signals through all channels using synchronous (rendezvous) communication. If, in the cooperative port, Spin happens to choose a rendezvous send statement that has no receiving part in the box process, the send statement is discarded by Spin, selecting a new candidate from the set of executable statements.

We verified the DFC box properties on the caller process, callee process, and four feature boxes that we have modelled. The verification of call waiting took the longest amount of time at 20 seconds with a maximum depth of 23938.

3.5.2 *Port Compliance*

Using synchronous communication with a cooperative environment to verify DFC-compliance implicitly assumes that the box only receives inputs and sends outputs when they are expected. To avoid the problem of having to verify ev-

ery box in the environment of every other box, we capture the essential behaviour of a DFC port in an abstract model of a port's behaviour. The DFC manual presents models of caller port and callee port behaviour [12]. We started using these caller and callee port models as our abstract ports, but found that the ports of the call waiting feature box can switch from being a caller to a callee and vice versa during the box's execution. This behaviour happens in two situations. The first situation occurs when a user who was *called* by the subscriber (interacting with CW through a caller port) is placed on hold. If this user decides to hang up, it releases the CW's caller port. The port just released can be used if another user *calls* the subscriber, and therefore interacts with CW through a callee port. The second situation occurs when a user *calls* the subscriber so the subscriber interacts with CW through a callee port, and another user tries to reach the subscriber. This user remains on hold. When the subscriber hangs up, the CW feature calls back the subscriber, reminding them that there is a person on hold. The subscriber is now *called* by CW and therefore interacts with CW through a caller port. Because of these situations, we created an abstract model of a port, called a *combo* port, that can switch between these modes. The details of how we verify all features with neighbours whose ports behave as this most general abstract port are provided in the next section. Because the abstract port is the union of the possible behaviours of every component's ports, false negatives are not possible. Each behaviour of the abstract port is the behaviour of some component's port. Next, we describe how to verify that the behaviour of every port of a box is contained within the possible behaviours of the combo port.

The abstract models of caller port, callee port, combo ports and their free and bound instances can be arranged in a partial order based on language containment as shown in Figure 7, where the dashed boxes are the abstract models (1-6). The language of the ports is the communication between the port and the channel. Bound ports (ports of bound boxes) have the behaviours of free ports, but after the call is torn down they return to their initial state to await another `setup` signal. The behaviour of both caller ports and callee ports is contained within the behaviour of the combo ports. The combo bound port model, shown in Figure 8, behaves like a caller port (or callee port) until it reaches the communication phase (state `CommPhase`), then there is no distinction in behaviour between caller port and callee port. This captures the behaviour of call waiting where the subscriber can call (using a call waiting callee port) or be called (using a call waiting caller port). Bound ports include the behaviour of the `busy` port, which handles the reception of a `setup` signal from the router when the box is already communicating with another box. The `busy` port rejects the request for an additional connection. This behaviour is captured by the looping transitions on states in Figure 8 labelled $\mathcal{U}$.

Rather than trying to show language containment of a box's port directly with the behaviour of the combo bound port, we rely on the partial order of abstract models, and match the box's ports with the most appropriate element of the abstract model hierarchy. This makes it easier to find the abstraction function needed to show language containment, and also provides a tighter verification of the port's behaviour. In Figure 7, the shaded boxes (the leaves of the tree) represent the ports of particular boxes. For example, the caller port of the caller process (7) is checked against the abstract bound caller port model (3), and the callee port of a free feature box (10) is checked against the free callee port abstract model (6).

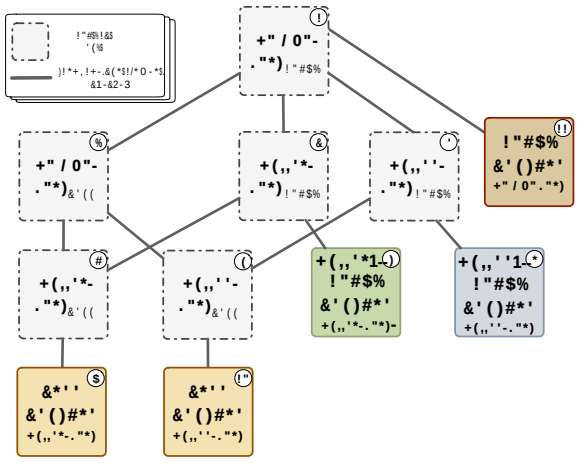

Figure 7: Abstract Models of Port Behaviour

We have written a simple tool that takes a description of the box as a set of transitions, isolates one port's behaviour, and carries out the port compliance check as explained in Section 2. It walks over transitions of a box, uses the abstraction function to compute the abstract states matching the source and destination of the transition, and checks that there is a corresponding transition in the abstract machine. The creation of the state transition diagram for a box is currently done by hand, but a tool could automatically extract these models from the PROMELA model. Determining the abstraction function is usually straightforward.

We checked the behaviour of all the boxes we modelled against the appropriate abstract port model. All the ports of call waiting behave as bound combo ports. We also checked the relationship between the abstract models (*e.g.,* that caller port bound is contained within the combo port bound). Using the partial order, we know that as long a box's port behaviour is contained within one of the abstract models, it is contained within the most general abstract model. This part of the verification effort took 5 seconds to check each call waiting port, which is the most complicated and time consuming example.

3.5.3 I/O Compliance

We use SPIN to verify the I/O compliance property: that each feature box placed in an environment of the most abstract ports (*i.e.,* communicating with combo bound ports) only receives signals it expects and sends only the signals the abstract port models expect. The environment for verifying a bound box with four communication ports, such as call waiting, is illustrated in Figure 9. Boxes in the transparent (T) category can have at most two ports, but user agents may have more. The channels `subsc` (subscriber), `ch1`, and `ch2` are for regular communication, and the `busy` channel handles the box's response to a fourth user that tries to call, where it simply tears down the call. By using

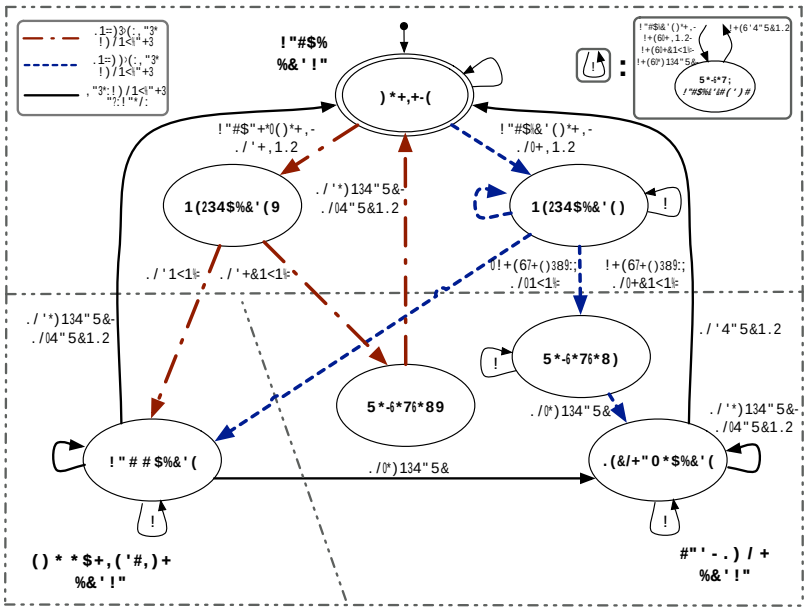

Figure 8: State Machine for $\texttt{ComboPort}_{\textsc{bound}}$ Process

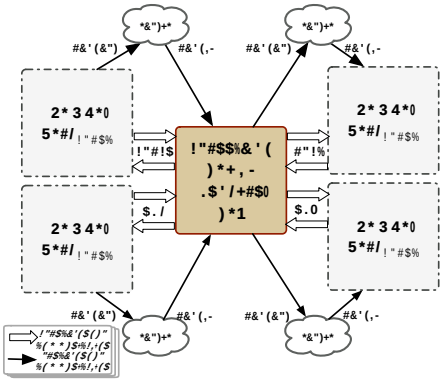

Figure 9: Abstract Environment to Verify Expected I/O Property

the combo bound port in the environment we capture the behaviour of multiple users communicating with a box as a caller or callee port at different times. The router process is included in the model, but the box is not dynamically created (as we are only checking one box) and the channels used are fixed in advance. As in our verification of fixed DFC configurations, we use synchronous communication for the channels `box_out` and `box_in`. We use asynchronous communication for the rest of the channels with a queue size of one. This forces the maximal amount of interleavings. Since we check that the box and its environment are free of deadlock, this channel size is sufficient.

To prove the I/O compliance property, we checked for a combination of lack of deadlock (invalid end states) and termination with empty queues. This is done automatically using XSpin. The maximum model checking time for checking the I/O compliance property for the CW box was 5 seconds.

3.5.4 *Inductive Reasoning*

In the final step of our compositional verification method, we use the DFC box properties stated in Table 2 to prove the segment properties, stated in Table 1, for all segments of an unknown, but finite length n. To gain confidence in the correctness of this proof, we verified it using the Hol theorem proving system [7].

Having shown that the DFC box properties hold for a box in any DFC environment, we can restrict our reasoning at this point to only rely on the LTL properties. We model the sending and receiving of signals at a box as uninterpreted predicates, Send and Receive, which take as parameters: time, the sender and receiver box addresses, and the signal. Since we are assuming the correctness of the routing protocol, the box address can be represented using natural numbers in the order they appear in the usage. Because we reason about segments, the base case is a *UUA* box connected to a *DUA* box communicating through a queue, whereas the inductive step is a proof for a *UUA* box connected to n transparent boxes and finalized by a *DUA* box to conclude a segment with $n + 1$ transparent boxes delimited by a *UUA* and a *DUA* (all communicating through queues).

Rather than using an embedding of temporal logic in the logic of the theorem prover, we found it easier to convert the segment properties to their equivalent versions as functions of time. For example, $\Box p$ means that proposition p holds at all points in the future, and is expressed using propositions as functions of time as $\forall n \cdot p(t + n)$, where t the time at which the property is supposed to hold.

Since we have completed the protocol compliance, I/O compliance and port compliance steps, we rely on the behaviour of DFC boxes, and can make the assumption that the queues behave perfectly meaning every signal sent is eventually received by the destination box. Using the DFC properties of boxes, plus the queue property, we prove the segment properties. For example, for segment property ***S*.1** describing

the propagation of `setup` signals, we prove,

$$
\begin{array}{l}
\quad \text{/* Queue Property (specialized for setup) */} \\
\quad (\forall \mathsf{c},\mathsf{t} \cdot (\mathsf{Send\ t\ c\ (c{+}1)\ Setup}) \Rightarrow \\
\qquad\quad (\exists \mathsf{t2} \cdot (\mathsf{t2} > \mathsf{t}) \wedge (\mathsf{Rec\ t2\ c\ (c{+}1)\ Setup}))) \\
\wedge \\
\quad \text{/* DFC Box Property T.2 */} \\
\quad (\forall \mathsf{c},\mathsf{t} \cdot (\mathsf{Rec\ t\ c\ (c{+}1)\ Setup}) \Rightarrow \\
\qquad\quad (\exists \mathsf{t2} \cdot (\mathsf{t2} > \mathsf{t}) \wedge (\mathsf{Send\ t2\ (c{+}1)\ (c{+}2)\ Setup}))) \\
\models \\
\quad \text{/* Segment Property S.1 */} \\
\quad \forall \mathsf{n},\mathsf{t} \cdot (\mathsf{Send\ t\ 0\ 1\ Setup}) \Rightarrow \\
\qquad (\exists \mathsf{t2} \cdot (\mathsf{t2} > \mathsf{t}) \wedge (\mathsf{Rec\ t2\ n\ (n{+}1)\ Setup}))
\end{array}
$$

where `t` and `t2` are variables of type "time" and `c` and `n` are addresses of boxes in the segment. At this level of reasoning, we also abstract away the details of which port a signal is sent on: the `Receive` predicate effectively means "there is some port on which the box receives this signal". The proofs of the propagation of the `setup` (***S*.1**), `avail` (***S*.4**), and `unavail` (***S*.5**) signals are all of similar form.

The proofs of the teardown properties (***S*.2**, ***S*.3**) are more complicated and involve a case split on whether a teardown signal is propagated or whether a teardown signal from the other end of the segment has been received and therefore the teardown is not propagated. The complete proofs are available in [13].

4. RELATED WORK

In this section, we briefly overview related work. There has been a variety of work on reducing a system consisting of a fixed number of finite-state processes communicating over unbounded FIFO channels to a completely finite state system for model checking (*e.g.,* [19], [18]). We use theorem proving in addition to model checking to handle an unknown number of components. While our theorem proving effort requires more work than model checking, it only needs to be completed once.

Most existing compositional reasoning techniques (*e.g.,* [5], [17], [15]), including assume-guarantee reasoning (*e.g.,* [8], [9], [1]) have focus on verifying systems with synchronous communication between a fixed number of heterogeneous components. We are interested in systems consisting of an unknown number of components with asynchronous communication. We have the advantage of the attribute of the semiregularity of the components with respect to the protocol allowing us to prove identical properties of all components and then use theorem proving to compose these results to show an overall system property.

Many systems, such as an IP-based implementation of DFC, have effectively unbounded queues because they send messages on the internet. Proving properties of systems with unbounded queues is undecidable in general [4]. In the future, we hope to build on work that characterizes decidable subsets of this problem (*e.g.,* [6]). Our use of theorem proving may offer us some advantages in being able to describe the required properties of queues abstractly.

There have been other efforts to verify DFC-related artifacts. Zave provided a formal description of the service layer of a telecommunication system, organized according to the DFC [20] architecture, using PROMELA and Z. The routing algorithm as well as the routing data were described in Z, and the DFC protocols were described in PROMELA. SPIN was used to check that the protocols of the virtual network never deadlock. As in Zave's work, the first step of our verification approach was to check for absence of deadlock, but we also check our call protocol properties for segments and provide a compositional approach for checking the properties on segments of unknown length.

The AT&T IP-based implementation of DFC, formerlly called ECLIPSE, was developed at AT&T Labs, and the Mocha model checker was used to verify the communication protocols [2],[3]. Individual ECLIPSE feature box code was translated to the modeling language framework of Mocha automatically. Similar to our work, the verification consists of combining a feature with standardized environmental peer entities of caller, callee and dual ports. They checked for deadlock using synchronous communication between the feature and its environment. The example described in their paper involves the analysis of a free transparent feature box only, and there is no discussion of analyzing bound feature boxes. We extend this work by checking liveness properties, as well taking the step of showing that all box behaviour is contained within an abstract model, which captures the most general DFC port behaviour. The dual port peer in the AT&T IP-based implementation of DFC work reflects the combined behaviour of a caller and callee ports, as described in the DFC manual. However, a dual port does not take into account busy processing, which is part of our abstract models. We also present a partial order among these abstract models. Finally, we also state box and segment properties and use inductive reasoning to conclude the segment properties.

5. CONCLUSION

In this work, we described a compositional reasoning method for protocol properties of port-based distributed systems with chains consisting of an unknown number of components and communicating using bounded queues. We demonstrated how the method can be used to verify liveness properties of the DFC call signalling protocol. Our verification method allows us to reason about components individually, which considerably reduces the verification effort. Our method consists of verifying an individual component for (1) individual properties in a cooperative environment with synchronous communication; (2) I/O compliance in an environment consisting of abstract models of port behaviour and asynchronous communication; and (3) port-compliance, where we show that the behaviour of every port in a component is within the behaviour of the abstract port. For DFC, we proved the language containment relationships between a partial order of abstract port models so that the most specific abstract port model could be used for checking the port compliance of a component. Finally, in a step that only needs to be completed once and is in terms of properties only, we prove by induction that the overall system properties hold for chains with an unknown number of components. This form of compositional reasoning is possible when a system with port-based communication has the attribute of semiregularity for the properties of interest, which is common for protocol properties. We expect the abstract ports and I/O compliance step to be reusable for multiple

types of properties of the system.

In the future, we plan to study the applicability of our method to other properties and other systems. Zave defined a set of constraints on feature behaviour called "ideal address translation" to evaluate and avoid undesirable interactions in DFC [24]. Adherence of every feature to the constraints results in provable properties such as preservation of anonymity. We plan to examine whether our method can be used to verify these properties of chains of features. We also plan to investigate at whether it is possible to determine automatically (to at least a first approximation) the properties that individual components must satisfy for the whole system to satisfy a given property, and whether an abstract model of a port can be determined automatically. Finally, we plan to study whether our method can be extended to work for properties of more complicated topologies than chains of components.

6. REFERENCES

[1] N. Amla, E. A. Emerson, and K. S. Namjoshi. Efficient decompositional model-checking for regular timing diagrams. In *CHARME*, 1999.

[2] G. Bond, F. Ivancić, N. Klarlund, and R. Trefler. ECLIPSE feature logic analysis. *IP-Telephony Workshop*, pages 49–56, 2001.

[3] G. W. Bond, E. Cheung, K. H. Purdy, P. Zave, and J. C. Ramming. An open architecture for next-generation telecommunication services. *ACM Transactions on Internet Technology*, 4(1):83–123, February 2004.

[4] D. Brand and P. Zafiropulo. On communicating finite-state machines. *J. ACM*, 30(2):323–342, 1983.

[5] J. Burch, E. Clarke, and D. Long. Symbolic model checking with partitioned transition relations. In *International Conference on Very Large Scale Integration*, pages 49–58, 1991. IFIP Transactions, North-Holland.

[6] X. Fu, T. Bultan, and J. Su. Conversation protocols: A formalism for specification and verification of reactive electronic services. In *Int'l Conf. on Impl. and Application of Automata*, number 2759 in LNCS, pages 188–200. Springer, 2003.

[7] M. J. C. Gordon and T. F. Melham. *Introduction to HOL: A theorem proving environment for higher order logic*. Cambridge University Press, 1993.

[8] O. Grumberg and D. E. Long. Model checking and modular verification. *ACM Trans. on Prog. Lang. and Sys.*, 16(3):843–871, 1994.

[9] T. Henzinger, S. Qadeer, and S. Rajamani. You assume, we guarantee: Methodology and case studies. In CAV, pages 440–451. 1998.

[10] G. J. Holzmann. *The Spin Model Checker: Primer and Reference Manual*. Addison-Wesley, 2003.

[11] M. Jackson and P. Zave. Distributed feature composition: A virtual architecture for telecommunications services. *IEEE Trans. on Soft. Eng.*, pages 831–847, August 1998.

[12] M. Jackson and P. Zave. *The DFC Manual*. AT&T Labs, November 2003.

[13] A. L. Juarez Dominguez. Verification of DFC call protocol correctness criteria. Master's thesis, School of Computer Science, University of Waterloo. May, 2005.

[14] K.Bhargavan, D. Obradovic, and C. A. Gunter. Formal verification of standards for distance vector routing protocols. *J. of the ACM*, 49(4):538–576, July 2002.

[15] K. L. McMillan. Verification of infinite state systems by compositional model checking. In *CHARME*, pages 219–233. 1999.

[16] A. Pnueli. In transition from global to modular temporal reasoning about programs. In *Logic and models of concurrent systems*, pages 123–144. 1985.

[17] H. J. Touati, H. Savoj, B. Lin, R. Brayton, and A. Sangiovanni-Vicentelli. Implicit state enumeration of finite state machines using BDD's. In *IEEE Int. Conf. Computer-Aided Design*, pages 130–133, 1990.

[18] A. Vardhan, K. Sen, M. Viswanathan, and G. Agha. Actively learning to verify safety for fifo automata. In *24th Conference on Foundations of Software Technology and Theoretical Computer Science* , pages 494–505, December 2004.

[19] A. Vardhan, K. Sen, M. Viswanathan, and G. Agha. Learning to verify safety properties. In *International Conference on Formal Engineering Methods (ICFEM'04)*, pages 274–289, 2004.

[20] P. Zave. Formal description of telecommunication services in Promela and Z. In *Proceedings of the 19th International NATO Summer School: Calculational System Design*, pages 395–420, 1999.

[21] P. Zave. Feature-oriented description, formal methods, and DFC. In *Language Constructs for Describing Features*, pages 11–26. Springer-Verlag, 2000/2001.

[22] P. Zave. An experiment in feature engineering. In *Monographs In Computer Science, Programming Methodology*, pages 353–377. Springer-Verlag, 2003.

[23] P. Zave. Ideal connection paths in DFC. Technical report, AT&T Laboratories–Research, November 2003.

[24] P. Zave. Address translation in telecommunication features. *ACM Trans. on Soft. Eng. and Methodology*, 13(1):1–36, Jan. 2004.

[25] P. Zave and M. Jackson. DFC modifications II: Protocol extensions. Technical report, AT&T Laboratories–Research, November 1999.

[26] P. Zave and M. Jackson. DFC modifications I: Routing extensions. Technical report, AT&T Laboratories–Research, May 2000.

[27] P. Zave and M. Jackson. A call abstraction for component coordination. In *Int. Coll. on Automata, Languages, and Programming: Workshop on Formal Methods and Component Interaction*, June 2002.