Analyzing Singularity Channel Contracts *

Zachary Stengel and Tevfik Bultan
Computer Science Department
University of California
Santa Barbara, CA 93106, USA
{zss,bultan}@cs.ucsb.edu

ABSTRACT

This paper presents techniques for analyzing channel contract specifications in Microsoft Research's Singularity operating system. A channel contract is a state machine that specifies the allowable interactions between a server and a client through an asynchronous communication channel. We show that, contrary to what is claimed in the Singularity documentation, processes that faithfully follow a channel contract can deadlock. We present a realizability analysis that can be used to identify channel contracts with problems. Our realizability analysis also leads to an efficient verification approach where properties about the interaction behavior can be verified without modeling the contents of communication channels. We analyzed more than 90 channel contracts from the Singularity code distribution and documentation. Only two contracts failed our realizability condition and these two contracts allow deadlocks. Our experimental results demonstrate that realizability analysis and verification of channel contracts can be done efficiently using our approach.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification—*formal methods, model checking*

General Terms

Verification

Keywords

realizability, conversations, asynchronous communication

1. INTRODUCTION

Singularity is a new, experimental, operating system developed by Microsoft Research to explore new approaches to OS design [11]. One of its main goals is to improve the dependability of software systems by rethinking some design decisions that have largely governed operating system architecture to date. Process isolation is a chief design principle of the Singularity operating system. To achieve this, certain constraints are enforced to ensure process independence. Among these is the rule that processes cannot share memory with each other or the kernel. All inter-process communication in Singularity, therefore, occurs via message passing over bidirectional conduits, called channels.

Channels have two end points referred to as the client and the server. The client and the server processes use the channel to communicate with each other by sending and receiving messages. Communication through Singularity channels correspond to asynchronous communication via FIFO queues. When a process sends a message through a channel, the message is appended to a message queue. A message that is at the head of a message queue is removed from the message queue when a receive action is executed by the receiving process at the other end of the channel.

In Singularity, each channel is governed by a channel contract [4, 15]. A channel contract is basically a state machine that specifies the allowable ordering of messages between the client and the server. Singularity processes are written in an extension of C# called Sing#, which provides constructs for writing channel contracts. The Sing# compiler statically checks that the processes that communicate through a channel conform to the channel contract. Singularity Design Note 5 [15] claims that client and server processes that are verified with respect to a channel contract are guaranteed not to deadlock. In this paper, we show that this is not correct. In fact, we show two Singularity channel contracts that allow deadlocks, one from the Singularity documentation and one from the Singularity code distribution.

The major contributions of this paper are: **(1)** We refine the realizability conditions from earlier work on multi-party conversation protocols [5], and apply them to channel contracts in the Singularity operating system. The new result we present in this paper is that autonomous condition is a sufficient condition for the realizability of two-party deterministic conversation protocols whereas the earlier results require two additional conditions to guarantee realizability of multi-party conversation protocols. Moreover, the autonomous condition is not directly applicable to channel contracts since all channel contracts violate the autonomous condition if the `ChannelClosed` messages are taken into account. In this paper, we show that the `ChannelClosed` messages can be ignored during realizability analysis due to the specific channel closing semantics of the Singularity channels. **(2)** We present a tool, called Tune, for analyzing the correctness of Singularity channel contracts. While Singularity's type checker provides conformance checks against contract implementations, our tool is able to verify the correctness of the contracts themselves, independent of any particular imple-

*This work is supported by NSF grants CCF-0614002 and CCF-0716095.

WOODSTOCK '97 El Paso, Texas USA

mentation. If potential problems are found, our tool utilizes the Spin model checker to generate a specific counter-example. In addition, our tool generates Promela models for Singularity contracts, which can be used in conjunction with Spin to verify application-specific LTL properties. **(3)** We experimentally evaluate the effectiveness of our approach by analyzing a large set of channel contracts from the Singularity code base. Our results demonstrate that our analysis techniques are efficient and are able to identify faulty channel contracts that allow deadlocks.

In this paper, we formalize the semantics of channel contracts and define the following realizability problem: Given a channel contract, is it possible to find client and server implementations that generate all the message sequences specified by the channel contract without causing any deadlock. Moreover, we present a sufficient condition for realizability of the channel contracts. The realizability condition simply states that all the transitions that originate in the same state should have the same sender. I.e., a send transition initiated by the client and a send transition initiated by the server cannot originate from the same state. This condition is sufficient for showing realizability; however, it is not a necessary condition: There are realizable channel contracts that violate this condition. Although our realizability analysis can generate false positives, our experiments on the channel contracts provided in the Singularity code base and documentation show that our realizability condition is not too restrictive. We analyzed more than 90 Singularity contracts and only two of them failed our realizability condition. And, the two contracts that failed our realizability condition have deadlocks, i.e., we did not observe any false positives in our experiments.

We implemented our approach in a tool called Tune. Sing# channel contracts are analyzed using Tune in three phases. In Phase 1 Tune simply checks the realizability condition. If the contract fails the realizability test, then we go to Phase 2. In Phase 2, Tune generates a Promela specification (input language of the Spin model checker [8]), and uses the Spin model checker to look for a deadlock. If Spin finds a deadlock, the message exchange sequence that leads to deadlock is reported to the user. Finally, in Phase 3 we use the Spin model checker to verify LTL properties about the channel contracts. These LTL formulas specify properties that are expected to hold for any possible message sequence generated by peers that behave according to the contract specification. Due to asynchronous communication, the state space can be unbounded (if the channels are unbounded). Since Spin is a finite state model checker, we have to bound the channel sizes. However, if the given channel contract satisfies our realizability condition, then it is not necessary to use asynchronous communication semantics: It is guaranteed that the behaviors generated by the synchronous and asynchronous communication semantics are equivalent as far as the messages sequences are concerned. Hence, we can verify the synchronous model and avoid the potential state space explosion due to channel contents. Our experiments show that Singularity contracts can be analyzed efficiently using our approach.

2. CHANNEL CONTRACTS

A Singularity channel consists of exactly two endpoints, referred to as peers. Channel endpoints are asymmetric, with one end being designated as the exporting end and the other designated the importing end. Messages sent over a channel are guaranteed to be received in FIFO order. For every contract `C`, a type is defined for interacting with each endpoint: `C.Exp` for the exporting endpoint, and `C.Imp` for the importing endpoint. An endpoint can only be owned by at most one thread at any time, which is responsible for dequeuing and processing messages sent to an endpoint [11]. We refer to the owner of the exporting endpoint as the server and the owner of the importing endpoint as the client. Process communication over Singularity channels is governed by a channel contract. A channel contract defines the following:

- The set of messages that may be transmitted over the channel.
- A finite state machine recognizing valid sequences of messages sent between two peers (client and server).

Figure 1 shows the basic syntax for contract state machine declarations.

The first state declaration in a contract is considered the initial state. Each state is defined by a series of message sequences. A message sequence begins with an action, which can be either sending a message from the server to the client or sending a message from the client to the server. Contracts are written from the perspective of the server; thus a message sent by the server is indicated by a `!` following the message name, and a message sent by the client is indicated by a `?` following the message name.

An action is followed by a continuation, which can be one of the following: 1) Empty, ending the message sequence; 2) A state followed by another continuation; 3) A message sequence; or 4) A choice among two or more message sequences. Continuations beginning with either a message sequence or a choice are effectively short-hand to avoid declaring an explicit state; in these cases, an implicit state is created as part of the contract protocol.

Either peer may at any time send a special `ChannelClosed` message, after which point, that peer can no longer send messages. However, the other peer may continue to send or receive messages according to the contract. Once both peers have sent the `ChannelClosed` message, the channel is fully closed, and this is the implicit final state of the channel contract. Also a state which defines no message sequences is considered an explicit final state. When a channel is in an explicit final state the only message which may be sent is the `ChannelClosed` message.

```
StateDeclaration ::= StateId : { MessageSequence* }
MessageSequence  ::= Action Continuation
Action ::= MessageId ! | MessageId ?
Continuation ::= ;
               | -> StateId Continuation
               | -> MessageSequence
               | -> Choice
Choice ::= ( MessageSequence (or MessageSequence)+ )
StateId ::= chars+
MessageId ::= chars+
```

Figure 1: Singularity Channel Contract Specification Syntax

```
public contract KeyboardDeviceContract {
    state Start: {
        Success! -> Ready;
    }
    state Ready: {
        GetKey? -> Waiting;
        PollKey? -> (AckKey! or NakKey!) -> Ready;
    }
    state Waiting: {
        AckKey! -> Ready;
        NakKey! -> Ready;
    }
}
```

Figure 2: An Example Channel Contract

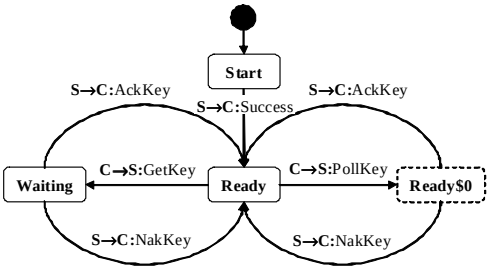


Figure 3: KeyboardDeviceContract State Machine

Figure 2 shows a simplified version of a contract governing a channel used by Singularity for interacting with a keyboard device. A full contract would also include explicit message declarations; however they are omitted here for simplicity. This contract defines three explicit states: `Start`, `Ready`, and `Waiting`, and also contains an implicit state as the target of the `PollKey?` action. Figure 3 shows the state machine that corresponds to the `KeyboardDeviceContract` channel contract. The implicit state is indicated by a dashed border. Also, in the Figure 3, we show the sender and the receiver of each message explicitly: **C** → **S** denotes a message sent by the client to the server, and **S** → **C** denotes a message sent by the server to the client.

This contract begins in the `Start` state and transitions into `Ready` state when the server sends a `Success` message. Once in the `Ready` state, the client may send either the `GetKey` message or the `PollKey` message. If the `GetKey` message is sent, the channel contract transitions to the `Waiting` state. If the `PollKey` message is sent, the contract transitions to an implicit state, named `Ready$0` in Figure 3. From either of these states, the server may send either the `AckKey` or `NakKey` message, which will transition the contract back to the `Ready` state. Note that the actions that correspond to sending of the `ChannelClosed` message and the implicit states created due to these actions are not shown in Figure 3.

Channel Closing: In Figure 4, we show the hierarchical state machine that corresponds to `KeyboardDeviceContract` with the implicit states for channel closing included. States `Start`, `Ready`, `Ready$0`, and `Waiting` are denoted as S, R, R0, and W in Figure 4. The hierarchical state machine shown in Figure 4 consists of 3 super-states (O,O), (C,O), (O,C), and one final state (C,C). State (C,C) indicates that both the client and server have closed the channel, whereas state (O,O) indicates that neither of them closed the channel yet. Similarly, state (C,O) indicates that only the client has closed the channel whereas state (O,C) indicates that only the server has closed the channel.

The four transitions among the three super-states and the state (C,C) correspond to multiple individual transitions among the sub-states. While in a substate of the super-state (O,O), if the client sends the `ChannelClosed` message, then the state machine transitions to the corresponding sub-state of the super-state (C,O). For example, from the state R(O,O), when the client sends the `ChannelClosed` message, the state machine goes to the state R(C,O). Similarly, while in a substate of the super-state (O,O), if the server sends the `ChannelClosed` message, then the state machine transitions to the corresponding sub-state of the super-state (O,C). Note that, this is not the standard semantics for the transitions in hierarchical state machine languages such as Statecharts, but it is what we need to model the channel closing semantics.

Neither peer is allowed to send any other message after sending

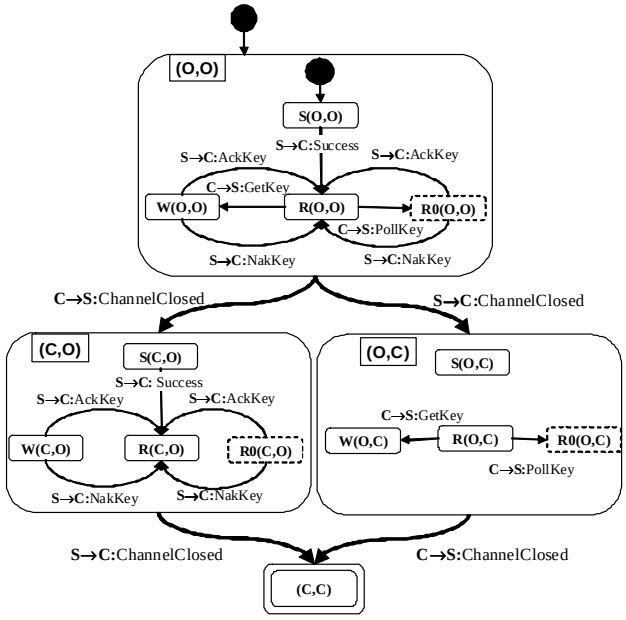


Figure 4: KeyboardDeviceContract with the Implicit States for Channel Closing shown as a Hierarchical State Machine

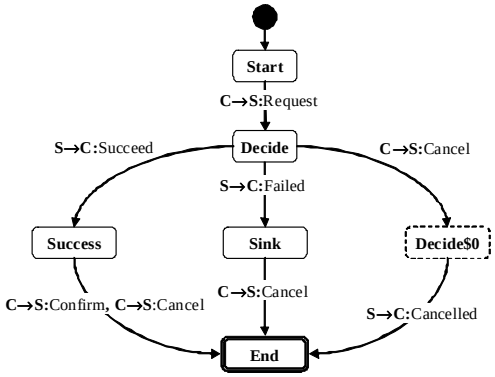


Figure 5: ReservationSession State Machine

the `ChannelClosed` message. So, in the sub-states of the super-state (C,O), there are no transitions where client is the sender, and in the sub-states of the super-state (O,C) there are no transitions where the server is the sender. However, after one peer closes the channel, the other peer can still send messages until it also closes the channel. When both peers close the channel we reach the final state (C,C).

2.1 Contracts with Problems

In the Singularity Design Note 5 [15], it is stated that

> *"clients and servers that have been verified separately against the same contract C are guaranteed not to deadlock when allowed to communicate according to C."*

The channel contract analysis we present in this paper uncovered two contracts which show this statement to be false. We discuss these two contracts below.

ReservationSession Contract: The first Singularity contract we discovered demonstrating the potential for deadlock is called the

Table 1: ReservationSession Deadlock Scenario

Server Action	Time Step	Client Action
	T0	**Send:** `Request`
Recv: `Request`	T1	
Send: `Succeeded`	T2	
	T3	**Send:** `Cancel`
Recv: `Cancel`	T4	

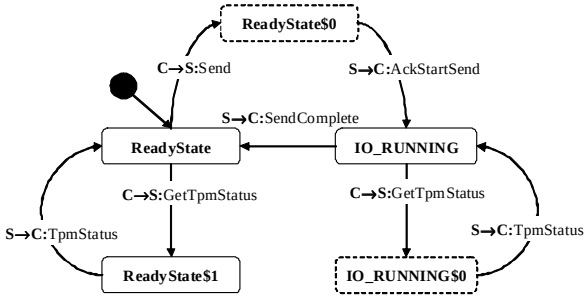


Figure 6: TpmContract State Machine

`ReservationSession` contract. It is an example contract given in the Singularity RDK documentation [15]. It defines the state machine shown in Figure 5. Table 1 shows an interleaving of valid client and server actions according to this contract that leads to a deadlock.

After step T4, the server is in the terminal `End` state. The client, however, is in the `Decide$0` state waiting for the server to send the `Cancelled` message. Neither peer can therefore make progress, and the channel is deadlocked.

Although this contract is given in the Singularity documentation, it is not included as code in the Singularity distribution. We created a user application containing this contract and added it to the distribution. The contract compiled and passed Singularity's static verification checks. Our application was allowed to run and we were able to witness the deadlock scenario, demonstrating that Singularity processes can deadlock even when they are faithfully following a channel contract.

TpmContract: The second contract we discovered violating the claim that Singularity contracts are deadlock-free, is the `TpmContract`. This contract is included in version 2.0 of the Singularity RDK and is used by a component of the Singularity kernel. Our analysis shows that it is possible for a client and server that obey this contract to deadlock. The state machine for this contract is shown in Figure 6.

Table 2 shows an interleaving of valid client and server actions that can lead to deadlock in this contract. After step T1, once the `Send` message is sent by the client and received by the server, both

Table 2: TpmContract Deadlock Scenario

Server Action	Time Step	Client Action
	T0	**Send:** `Send`
Recv: `Send`	T1	
Send: `AckStartSend`	T2	
	T3	**Recv:** `ACkStartSend`
Send: `SendComplete`	T4	
	T5	**Send:** `GetTpmStatus`
Recv: `GetTpmStatus`	T6	
Send: `TpmStatus`	T7	

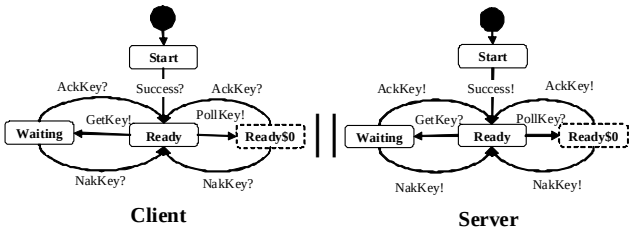


Figure 7: Projection of the KeyboardDeviceContract to the Client and the Server

peers are in the `ReadState$0` state. After step T3, both peers are in the `IO_RUNNING` state. After step T4, the server transitions to the `ReadyState` state. However, after step T5, the client transitions to the `IO_RUNNING$0` state. Deadlock is now inevitable. At step T6, the server receives the `GetTpmStatus` message and responds, at step T7, by sending the `TpmStatus` message. The server is in the `ReadyState` state waiting for the client to send the `Send` or `GetTpmStatus` messages; however, the client cannot make progress because the `SendComplete` message is at the head of its receive queue, for which there is no valid transition. The channel is therefore deadlocked. Note that, although the problems with both of these contracts involve transitions to implicit states, there is no difference between implicit and explicit states with respect to this type of problems. The equivalent contracts with only explicit states will exhibit the same problems.

Here is an interesting distinction between the channel contract shown in Figure 3 and the channel contracts shown in Figures 5, and 6: If a client and a server implement the contract in 3 by simply using the projections of the contract machine (where a send action for the server becomes a receive action for the client and visa versa), then the resulting system will follow the contract and will not deadlock. In Figure 7, we visually show the projection of the `KeyboardDeviceContract` to the client and the server. However, if we do the same for the `ReservationSession` contract shown in Figure 5 and the `TpmContract` contract shown in Figure 6, the resulting system can deadlock. Below, we formalize this difference as the realizability problem for channel contracts: `KeyboardDeviceContract` is realizable whereas `ReservationSession` and `TpmContract` are not. Moreover, we give a sufficient condition for the realizability of channel contracts.

3. A FORMAL MODEL

A *channel contract automaton* is a tuple $A = (M, S, I, F, \delta)$ where M is the finite set of messages (the alphabet), S is the finite set of states, $I \in S$ is the initial state, $F \subseteq T$ is the set of final states, and $\delta \subseteq S \times M \times S$ is a deterministic transition relation, i.e., $(s, m, s') \in \delta$ and $(s, m, s'') \in \delta$ implies that $s' = s''$. (Since Singularity does not allow multiple transitions from the same contract state with the same message label, the transition relations of the channel contract automata are deterministic.)

A channel contract automaton recognizes sequences of messages, i.e., members of the set M^*. We call each such message sequence a *conversation*. The language accepted by a channel contract automaton A is denoted by $L(A) \subseteq M^*$, and it specifies the set of allowable conversations (i.e., the set of allowable message sequences) between a client and a server according to the channel contract.

The set of messages is partitioned to two sets, $M = M_c \cup M_s$, where M_c is the set of messages that the client can send to the server and M_s is the set of messages that the server can send to

the client. In order to simplify our technical model we assume that $M_c \cap M_s = \emptyset$. This does not reduce the generality of our model. If there are messages that both peers can send (for example, the `ChannelClosed` message), we add the sender's initial to such messages to identify the sending peer (for example `CChannelClosed` if the client is the sender and `SChannelClosed` if the server is the sender).

Given a channel contract and the corresponding channel contract automaton, a server and a client conform to the contract if the sequence of messages sent by the server and the client (recorded in the order they are sent) is a conversation that is accepted by the channel contract automaton. We also require that 1) the peers do not deadlock and 2) all the messages that are sent are eventually consumed. We say that a server and a client realize a channel contract if they conform to that contract and, additionally, they can generate every conversation that is specified by the contract. We will formalize these definitions below, after discussing the semantics of channel closing.

Channel Closing Semantics: Given a channel contract automaton $A = (M, S, I, F, \delta)$, we model the channel closing semantics by partitioning the set of states to four sets: $S = S_{o,o} \cup S_{o,c} \cup S_{c,o} \cup S_{c,c}$, where $|S_{o,o}| = |S_{c,o}| = |S_{o,c}|$ and $|S_{c,c}| = 1$. States in the set $S_{o,o}$ correspond to states before any `ChannelClosed` messages have been sent. States in the set $S_{c,o}$ correspond to states where the client has sent a `ChannelClosed` message but the server has not. States in the set $S_{o,c}$ correspond to states where the server has sent a `ChannelClosed` message but the client has not. Finally, $S_{c,c}$ contains a single state which corresponds to the state where both the client and the server have sent the `ChannelClosed` message, therefore, the channel is closed, denoting the end of the conversation. We define the final state as the state after both peers have sent the `ChannelClosed` message, hence, $F = S_{c,c}$.

Given a channel contract automaton $A = (M, S, I, F, \delta)$ and $S = S_{o,o} \cup S_{c,o} \cup S_{o,c} \cup S_{c,c}$, there exist two functions $f_{c,o} : S_{o,o} \rightarrow S_{c,o}$, and $f_{o,c} : S_{o,o} \rightarrow S_{o,c}$ such that,

- $f_{c,o}$ and $f_{o,c}$ are bijections.
- For each $s \in S_{o,o}$, $(s, \texttt{CChannelClosed}, f_{c,o}(s)) \in \delta$.
- For each $s \in S_{o,o}$, $(s, \texttt{SChannelClosed}, f_{o,c}(s)) \in \delta$
- For any $s, s' \in S_{o,o}$, $(f_{c,o}(s), m, f_{c,o}(s')) \in \delta$ if and only if $(s, m, s') \in \delta$ and $m \in M_s$.
- For any $s, s' \in S_{o,o}$, $(f_{o,c}(s), m, f_{o,c}(s')) \in \delta$ if and only if $(s, m, s') \in \delta$ and $m \in M_c$.

Finally, we have the following three conditions:

- For each $s \in S_{c,o}$, $(s, \texttt{SChannelClosed}, s') \in \delta$ where $s' \in S_{c,c} = F$.
- For each $s \in S_{o,c}$, $(s, \texttt{CChannelClosed}, s') \in \delta$ where $s' \in S_{c,c} = F$.
- There are no transitions that originate in $S_{c,c}$, i.e., after both peers close the channel no more messages can be sent.

3.1 Execution Model

In order to formally define the conformance and realizability for channel contracts, we have to define an execution model that characterizes the behaviors of the client and the server processes and formally defines the semantics of asynchronous communication via unbounded communication channels.

An *asynchronous channel system* $P_c \parallel P_s$ consists of two state machines: the client state machine $P_c = (M, S_c, I_c, F_c, \delta_c)$ and the server state machine $P_s = (M, S_s, I_s, F_s, \delta_s)$ where M is the finite set of messages, S_c and S_s are the finite sets of states, $I_c \in S_c$ and $I_s \in S_s$ are the initial states, $F_c \subseteq S_c$ and $F_s \subseteq S_s$ are the sets of final states, $\delta_c \subseteq S_c \times M \times S_c$ and $\delta_s \subseteq S_s \times M \times S_s$ are the transition relations for the client and server machines, respectively, and δ_c and δ_s are both deterministic transition relations.

A *configuration* of a client-server system is a tuple of the form (Q_c, s_c, Q_s, s_s) where $s_c \in S_c$ denotes the state and $Q_c \in M_s^*$ denotes the contents of the incoming message queue of the client, $s_s \in S_s$ denotes the state and $Q_s \in M_c^*$ denotes the contents of the incoming message queue of the server.

For two configurations $\sigma = (Q_c, s_c, Q_s, s_s)$ and $\sigma' = (Q'_c, s'_c, Q'_s, s'_s)$, we say that σ *derives* σ', written as $\sigma \rightarrow \sigma'$, if one of the following conditions hold:

- Client sends a message (denoted as $\sigma \overset{C!m}{\rightarrow} \sigma'$) where $m \in M_c$, $(s_c, m, s'_c) \in \delta_c$, $Q'_s = Q_s m$, $Q'_c = Q_c$, and $s'_s = s_s$.
- Server sends a message (denoted as $\sigma \overset{S!m}{\rightarrow} \sigma'$) where $m \in M_s$, $(s_s, m, s'_s) \in \delta_s$, $Q'_c = Q_c m$, $Q'_s = Q_s$, and $s'_c = s_c$.
- Client receives a message (denoted as $\sigma \overset{C?m}{\rightarrow} \sigma'$) where $m \in M_s$, $(s_c, m, s'_c) \in \delta_c$, $Q_c = m Q'_c$, $Q'_s = Q_s$, and $s'_s = s_s$.
- Server receives a message (denoted as $\sigma \overset{S?m}{\rightarrow} \sigma'$) where $m \in M_c$, $(s_s, m, s'_s) \in \delta_s$, $Q_s = m Q'_s$, $Q'_c = Q_c$, and $s'_c = s_c$.

We use $\rightarrow^*$ to denote the reflexive transitive closure of the derivation relation, and we say that σ' is reachable from σ if $\sigma \rightarrow^* \sigma'$.

A run of a channel system is a sequence of configurations $\gamma = \sigma_0 \sigma_1 \ldots \sigma_k$, such that,

1. $\sigma_0 = (\epsilon, I_c, \epsilon, I_s)$,
2. $\sigma_j \rightarrow \sigma_{j+1}$ for $0 \leq j < k$, and
3. $\sigma_k = (\epsilon, s_c, \epsilon, s_s)$, $s_c \in F_c$ and $s_s \in F_s$.

If γ does not satisfy the third condition we call it a partial run.

We say that a channel system $P_c \parallel P_s$ is *well-behaved* if $(\epsilon, I_c, \epsilon, I_s) \rightarrow^* \sigma$ implies $\sigma \rightarrow^* (\epsilon, s_c, \epsilon, s_s)$, where $s_c \in F_c$ and $s_s \in F_s$. I.e., a channel system is well-behaved if every configuration reachable from the initial configuration can reach the final configuration.

Given a run γ, the conversation generated by γ, denoted as $conv(\gamma)$, is defined recursively as follows [7]:

- If $|\gamma| \leq 1$, then $conv(\gamma)$ is the empty sequence.
- If $\gamma = \gamma' \sigma \sigma'$, then
  - $conv(\gamma) = conv(\gamma' \sigma) m$ if $\sigma \overset{C!m}{\rightarrow} \sigma'$ or $\sigma \overset{S!m}{\rightarrow} \sigma'$
  - $conv(\gamma) = conv(\gamma' \sigma)$ otherwise.

A message sequence $w \in M^*$ is a *conversation* generated by a channel system $P_c \parallel P_s$ if there exists a complete run γ of $P_c \parallel P_s$ such that $w = conv(\gamma)$. We denote the set of all conversations generated by a channel system $P_c \parallel P_s$ as $conv(P_s \parallel P_s)$. Given a channel contract automaton A and a channel system $P_c \parallel P_s$:

- $P_c \parallel P_s$ *conforms to* A if $P_c \parallel P_s$ is well-behaved and $conv(P_s \parallel P_s) \subseteq L(A)$,
- $P_c \parallel P_s$ *realizes* A if $P_c \parallel P_s$ is well-behaved and $conv(P_s \parallel P_s) = L(A)$.

Given a contract automaton A, it is easy to generate a channel system that conforms to A. All the client and server have to do is to send the `ChannelClosed` messages to each other (and receive them). Since this simple conversation is part of any channel contract, the resulting channel system will conform to any channel contract. However, it is not easy to determine if there exists a channel system that realizes a contract automaton. In fact, this is the problem with the two example contracts (`ReservationSession` and `TpmContract`) we discussed earlier. Any channel system that tries to realize these two contracts can potentially deadlock, i.e., these channel contracts are not realizable. Next, we present a simple condition that guarantees realizability of channel contracts.

4. REALIZABILITY ANALYSIS

The realizability condition we propose is the following: If we ignore the `ChannelClosed` messages, then there should not be two transitions from the same state of the contract automaton such that the client is the sender on one of them and the server is the sender on the other one. Either the client should be the sender on all transitions from a given state or the server should be the sender. This condition is called the "autonomous" condition [5, 7]. Formally, a contract automaton $A = (M, S, I, F, \delta)$ is called *autonomous* if it satisfies the following condition

- for all $s \in S_{o,o}$, if there exists a transition $(s, m, s') \in \delta$ where $m \in M_c \backslash \{\texttt{CChannelClosed}\}$, then for all $(s, m', s'') \in \delta$, $m' \in M_c \cup \{\texttt{SChannelClosed}\}$, and
- for all $s \in S_{o,o}$, if there exists a transition $(s, m, s') \in \delta$ where $m \in M_s \backslash \{\texttt{SChannelClosed}\}$, then for all $(s, m', s'') \in \delta$, $m' \in M_s \cup \{\texttt{CChannelClosed}\}$.

Note that this condition can be checked in linear time in the number of transitions by simply traversing the transitions of the contract machine.

We will show that if a contract automaton is autonomous, then it is realizable. In fact, the contract automaton is realizable by its projection to the client and the server. During projection, a transition in the contract automaton that is labeled with a message that the server sends becomes a send transition for the server and a receive transition for the client. A transition labeled with a message that the client sends becomes a receive transition for the server and a send transition for the client. Figure 7 shows the projection of the `KeyboardDeviceConstract` in Figure 3 to the client and the server.

Formally, given a contract automaton $A = (M, S, I, F, \delta)$, let $\pi_c(A)$ denote its projection to the client and $\pi_s(A)$ denote its projection to the server: $\pi_c(A) = (M, S_c, I_c, F_c, \delta_c)$ where $S_c = S$, $I_c = I$, $F_c = F$, and $\delta_c = \delta$, and $\pi_s(A) = (M, S_s, I_s, F_s, \delta_s)$ where $S_s = S$, $I_s = I$, $F_s = F$, and $\delta_s = \delta$ (i.e., the client and server machines are identical with the contract automaton, the only difference is the interpretation of the transitions). Here is our main realizability result:

THEOREM 1. *Given a contract automaton A, if A is autonomous, then $\pi_c(A) \parallel \pi_s(A)$ realizes A.*

1) $\pi_c(A) \parallel \pi_s(A)$ is well-behaved: Given an autonomous contract automaton $A = (M, S, I, F, \delta)$, and the channel system $\pi_c(A) \parallel \pi_s(A)$, for every configuration $\sigma = (Q_c, s_c, Q_s, s_s)$ where $s_c \in S_{o,o}$ and $s_s \in S_{o,o}$ and σ is reachable from the initial configuration $(\epsilon, I, \epsilon, I)$, the following holds: Either $Q_c = \epsilon$ and the configuration $\sigma' = (\epsilon, s_c, \epsilon, s_c)$ is reachable from σ by only executing receive transitions by the server, or $Q_s = \epsilon$ and the configuration $\sigma' = (\epsilon, s_s, \epsilon, s_s)$ is reachable from σ by only executing receive transitions by the client. This can be proved by induction on the length of the run γ that reaches σ. Due to autonomous condition, either all the transitions from the initial state are send transitions by the server or they are all send transitions by the client. Without loss of generality, assume that it is the server who sends the first message. Then, the only transition that the client can take is to receive that message. If the server has another send transition from its new state, then it can send another message before client receives the first message. However, the client will not be able to send a message before it receives all the messages sent by the server. Moreover, the client can only reach a state where it has a send transition after receiving all the messages sent by the server (since the client and the server have identical deterministic state machines). At that point the server must be waiting at the same state since all the transitions from that state are receive transitions for the server and it has to wait for the client to send a message before it can proceed. The autonomous condition basically guarantees that the client and the server run in a token-passing style where they alternate between message sending and message receiving phases. Before each send action, each process receives all the messages in its incoming message queue, i.e., the incoming message queue of a process that is sending a message is always empty. When both queues are empty, both processes are in the same contract state.

In the above discussion we only considered the states before any `ChannelClosed` message has been sent. Assume that client sends the first `ChannelClosed` message. If the client sends the `ChannelClosed` message when its channel queue is empty, then based on the above property, the server can execute a series of receive transitions that terminates by receiving the `ChannelClosed` message. After that, the server is the only peer that can send messages, and the client can only receive the messages sent by the server. The server at any moment can send the `ChannelClosed` message and when the client receives that message, both peers will be in the final state and the message queues will be empty.

If the client sends the `ChannelClosed` message when its channel queue is not empty, then based on the above property, the client can continue to execute a series of receive transitions. When the server receives the `ChannelClosed` message sent by the client and when the client receives all the messages in its incoming message queue, both peers will be in the same state. They can both reach the final state when the server sends the `ChannelClosed` message and the client receives it. Symmetric arguments hold for the cases when the server sends the first `ChannelClosed` message.

2) $conv(\pi_c(A) \parallel \pi_s(A)) = L(A)$: It is relatively easy to show that $L(A) \subseteq conv(\pi_c(A) \parallel \pi_s(A))$. For any conversation recognized by A, we can simulate the execution of A by $\pi_c(A) \parallel \pi_s(A)$ by executing each receive action immediately after the corresponding send action. I.e., at any configuration there is at most one queue that is not empty and it holds at most one message. We call such a run an immediately receiving run. In order to show that $conv(\pi_c(A) \parallel \pi_s(A)) \subseteq L(A)$, we argue that, if A is autonomous, then for any run γ of $\pi_c(A) \parallel \pi_s(A)$, there exists another run γ' of $\pi_c(A) \parallel \pi_s(A)$ such that γ' is an immediately receiving run and $conv(\gamma') = conv(\gamma)$. Since the execution of $\pi_c(A) \parallel \pi_s(A)$ goes through phases where in each phase only one peer is sending messages, we can rearrange the receive actions in each such phase by moving each receive action right after the corresponding send. This will convert the original run to an immediately receiving run without changing the conversation.

Note that an immediately receiving run is equivalent to executing the client and the server with the synchronous communication semantics where the send and the receive actions occur together and the message queues are not used. The set of conversations gener-

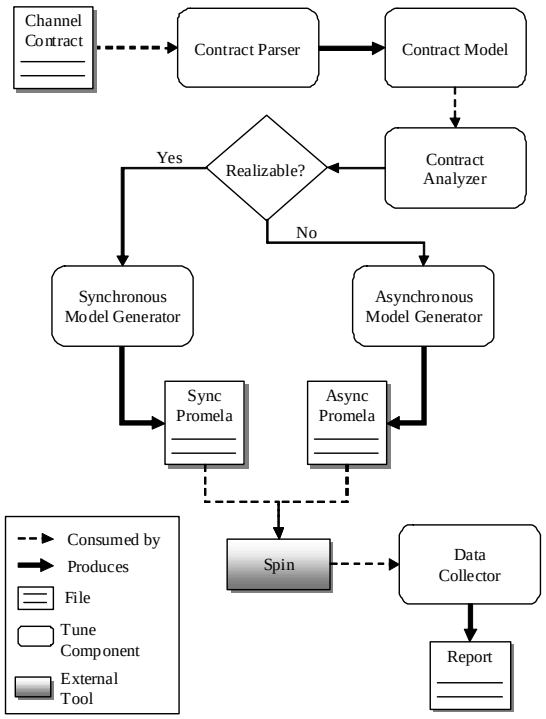


Figure 8: Tune Architecture

ated by the synchronous composition of the client and the server is equal to the conversations generated by the runs with immediate receives. Finally, note that, the synchronous composition of $\pi_c(A)$ and $\pi_s(A)$ is identical to A itself (synchronous composition simply combines the send and receive transitions in the peer machines that correspond to the projection of the same transition in A). Hence, we conclude that $conv(\pi_c(A) \parallel \pi_s(A)) \subseteq L(A)$.

LTL Model Checking: In addition to checking the realizability of a channel contract, we may also want to check properties about the message sequences allowed by the channel contract. We use LTL formulas to specify properties about the conversation set of a client-server channel system or a channel contract. During LTL verification we only investigate the behaviors before any `ChannelClosed` message is sent, i.e., we only model the states in $S_{o,o}$. We use the results of the realizability analysis to improve the efficiency of LTL verification. Since realizability of a channel contract implies that $conv(\pi_c(A) \parallel \pi_s(A)) = L(A)$, we use A instead of $\pi_c(A) \parallel \pi_s(A)$ during verification of LTL properties about message sequences.

5. TUNE: A TOOL FOR ANALYZING SING# CHANNEL CONTRACTS

We developed a tool called Tune for analyzing Singularity channel contracts written in Sing#. In addition to checking realizability of channel contracts, Tune produces representations of the contract in Promela (input language of the Spin model checker [8]) in order to detect deadlock traces and for LTL verification.

Figure 8 shows Tune's core architecture. Tune's Contract Parser recognizes a valid Sing# contract based on the specification defined in the Singularity Design Note 5 [15], with the following limitations:

```
typedef Endpoint
{
    chan Send = [ChannelSize] of { mtype };
    chan Recv = [ChannelSize] of { mtype };
}
typedef Channel
{
    Endpoint Imp;
    Endpoint Exp;
}
```

Figure 9: Endpoint and Channel declarations in Promela

1. Tune requires all messages specify explicit directional qualifiers. Messages must be sent either from server to client, or from client to server.
2. Tune does not support subroutine states.
3. Tune does not process message parameters.
4. Tune does not support bound variables.

In Sing# contract specifications message parameters do not influence the contract behavior, so we are able to ignore them during our analysis. The other three limitations (1, 2 and 4) do not change the expressiveness of the contract language, i.e., any contract using these features can be converted to an equivalent contract without them. In practice, we have found these limitations do not significantly impact our ability to analyze contracts implemented in the Singularity code base. 93 out of the 94 contracts included in version 1.1 of the RDK, and 93 of the 95 contracts in version 2.0 are fully recognized and supported by Tune.

Given a contract, the Contract Parser produces a state machine model representing the contract. This model is then submitted to the contract analyzer which checks the realizability condition we presented earlier. Depending on the results of this determination, the contract model is submitted to one of two model generators, which create Promela models representing the contract for LTL verification and deadlock trace generation using the Spin model checker. The results produced by Spin are then collected by the Data Collector, parsed, and used to produce a report containing a pass/fail result of verification along with analysis statistics and an error trace if the verification fails.

Asynchronous Promela Model: The Asynchronous Model Generator is responsible for creating a Promela model of a channel contract that allows for messages to be sent asynchronously via FIFO message queues between client and server processes. This enables the exhaustive exploration of all possible message sequences generated using a given message queue size. Like Singularity, Promela has a built-in channel construct for passing messages between independent and concurrently running processes. However, there are significant differences between channels in Singularity and channels in Promela which require special consideration when modeling Singularity channels in Promela. These differences are due primarily to the fact that Promela does not have a concept of a channel endpoint. In Promela, messages sent on a channel are received nondeterministically by one of any number of listeners on the channel for that message. However in Singularity, a channel has at most two processes (client and server) that are communicating on it. Singularity endpoints also introduce directionality constraints on messages whereas, in Promela, messages can always be sent in any direction.

Promela's `typedef` keyword enables the basic structural modeling of Singularity endpoints and channels. To begin the model,

```
proctype Server( Endpoint exp ) {
  m_serverIsRunning = true;
  /* Wait for the client to be ready to receive */
  m_clientIsRunning -> goto START;
START:
  exp.Send!OMSG_Success -> goto READY;
READY:
  /* Non-deterministically choose message sequence */
  if
  :: exp.Recv?[IMSG_GetKey] -> exp.Recv?IMSG_GetKey;
     goto WAITING;
  :: exp.Recv?[IMSG_PollKey] -> exp.Recv?IMSG_PollKey;
    /* Implicit Ready$0 state */
    if
    :: exp.Send!OMSG_AckKey -> goto READY;
    :: exp.Send!OMSG_NakKey -> goto READY;
    fi;
    goto READY;
  fi;
WAITING:
  if
  :: exp.Send!OMSG_AckKey -> goto READY;
  :: exp.Send!OMSG_NakKey -> goto READY;
  fi;
}
```

Figure 10: KeyboardDeviceContract Server proctype

```
proctype Client( Endpoint imp ) {
  m_clientIsRunning = true;
  goto START;
START:
  imp.Recv?OMSG_Success -> goto READY;
READY:
  /* Non-deterministically choose message sequence */
  if
  :: imp.Send!IMSG_GetKey -> goto WAITING;
  :: imp.Send!IMSG_PollKey ->
     /* Implicit Ready$0 state */
     if
     :: imp.Recv?[OMSG_AckKey] -> imp.Recv?OMSG_AckKey;
        goto READY;
     :: imp.Recv?[OMSG_NakKey] -> imp.Recv?OMSG_NakKey;
        goto READY;
     fi;
     goto READY;
  fi;
WAITING:
  /* Non-deterministically choose message sequence */
  if
  :: imp.Recv?[OMSG_AckKey] -> imp.Recv?OMSG_AckKey;
     goto READY;
  :: imp.Recv?[OMSG_NakKey] -> imp.Recv?OMSG_NakKey;
     goto READY;
  fi;
}
```

Figure 11: KeyboardDeviceContract Client proctype

an `Endpoint typedef` is declared containing two Promela channels, one for sending and one for receiving. A `Channel typedef` is also declared, consisting of two `Endpoint` declarations, one called `exp` (for the server), and the other called `imp` (for the client). Figure 9 shows the `typedef` declarations representing a Singularity endpoint and channel respectively. Promela channels must be bounded, therefore a configurable `ChannelSize` variable is used to bound the size of the `Send` and `Recv` endpoint channels. Contract messages and states are declared as enumerations using Promela's `mtype` construct. The prefixes `OMSG_` and `IMSG_` denote messages sent by the server and the client, respectively, and the prefix `STATE_` is used to identify the states.

Channel contract verification is accomplished using two processes, declared using Promela's `proctype` construct, one representing the server and one representing the client. The client and server processes are projections of the contract state machine. Since the contract is written from the perspective of the server, the server process directly follows the contract state machine, while for the client process message directions are reversed (i.e., sends become receives and receives become sends). In each state, each process nondeterministically chooses one of the enabled send or receive actions. Implicit states are not explicitly represented in the client and server processes, but are instead represented as nested `if` blocks within their preceding explicit state. Figures 10 and 11 show the Promela models representing the server and client processes for the `KeyboardDeviceContract` described in Section 2.

In addition to the server and client processes, a third process is also declared, which represents the Singularity channel itself. This process is responsible primarily for contract monitoring. It polls the `Send` channel of both the client and server `Endpoint` structures. Once a message becomes available, it is atomically retrieved from the `Send` channel queue, validated against the contract specification, and redelivered on the `Recv` channel of the peer's `Endpoint`. The atomicity of these steps guarantee the in-order delivery of messages from one process to the other. To allow for granular contract monitoring, implicit states are represented explicitly in the contract process.

Using these three processes, Spin will exhaustively explore all possible interleavings of send and receive actions. Verification is accomplished by defining LTL properties representing correctness criteria, which should never be violated by a valid contract. By default, Tune defines the following LTL formula for all contracts: `G(!timeout)` where `timeout` is a built-in Spin variable that is false unless no process can make progress. This property, therefore, enables us to test for deadlocks.

Synchronous Promela Model: The exhaustive explicit state verification performed against asynchronous contract models is, in general, unsound due to the fact that analysis can cause exponential state space explosion, and Promela channel queues must have a fixed bound. A sound and more efficient alternative is to first perform realizability analysis against the contract. If a contract is determined to be realizable, then it is guaranteed to be deadlock-free and can be verified using an equivalent synchronous model. Tune performs realizability analysis against the contract model produced by the parser, and if the contract is determined to be realizable, the Synchronous Model Generator is used to create a synchronous Promela model representing the contract. Figure 12 shows the entire synchronous model for the `KeyboardDeviceContract`.

6. EXPERIMENTS

We developed a three phase approach for the verification of Singularity channel contracts. The first two phases together detect potential deadlocks in the channel contracts, and the third phase checks contract specific LTL properties to verify correctness.

Phase 1: Realizability Analysis: In the first phase we perform realizability analysis against the contract. If the contract is realizable, then it is guaranteed to be deadlock-free. Our realizability analysis is sound, however, it is conservative and may produce false positives. For example, Figure 13 shows a version of the `TpmContract` state machine described in section 3.2, which has been fixed to no longer contain a deadlock. The `IO_RUNNING$1` state and dashed transitions have been added.

The fixed `TpmContract` will not deadlock, however, the `IO_RUNNING` state still violates the autonomous property, and thus will fail our realizability check. Hence, if a contract fails realizability analysis, it may or may not contain a deadlock, and this

```
mtype = {Ready, Ready__1, Start, Waiting};
mtype = {m_GetKey, m_PollKey, m_AckKey, m_NakKey, m_Success};
mtype state;
mtype msg;
active proctype GuardedProtocol(){
  state = Start;
  msg= m_undefined;
  do
  :: if
     :: state==Ready ->
         d_step{ msg=m_GetKey; state=Waiting };
     :: state==Ready ->
        d_step{ msg=m_PollKey; state=Ready__1};
     :: state==Ready__1 ->
        d_step{ msg=m_AckKey; state=Ready};
     :: state==Ready__1 ->
        d_step{ msg=m_NakKey; state=Ready};
     :: state==Start ->
        d_step{ msg=m_Success; state=Ready};
     :: state==Waiting ->
        d_step{ msg=m_AckKey; state=Ready};
     :: state==Waiting ->
        d_step{ msg=m_NakKey; state=Ready};
    fi;
  od;
}
```

Figure 12: Synchronous KeyboardDeviceContract Promela Model

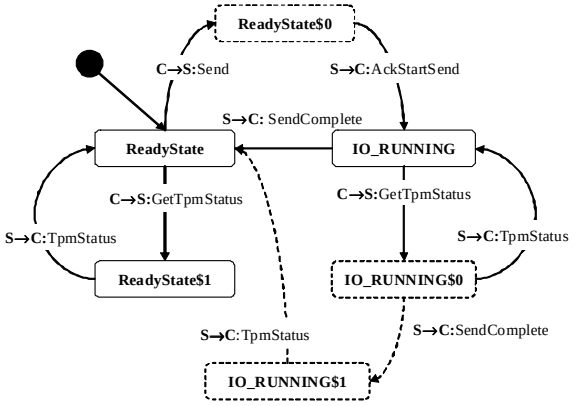


Figure 13: Fixed TpmContract State Machine

motivates the second phase of our analysis.

Phase 2: Deadlock Trace Generation: In phase two of our analysis, we run Spin against the full asynchronous Promela model to exhaustively search for deadlock conditions. This phase is required only for contracts that fail the realizability analysis in phase 1. The analysis in phase 2 produces no false positives—if it finds a deadlock, a counter-example trace is generated showing a precise message sequence that causes deadlock. Since channel sizes in Spin are bounded, this analysis is unsound in general, i.e., if we do not find a deadlock trace a deadlock may still exist for a system with larger message queues. Also, for bounded channels, the complexity of this analysis is exponential in the size of the channel in the worst case. Increasing the channel size (i.e. the length of the send and receive queues) can cause exponential growth in the state space. This is true even for very simple contracts, as demonstrated by the state machine shown in Figure 14.

Figure 15 shows the results of checking the following LTL formula: `G(F(m1) \/ F(m2))` against the asynchronous Promela model compared to the results of checking the same formula against

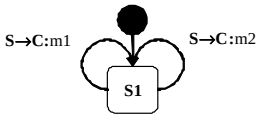
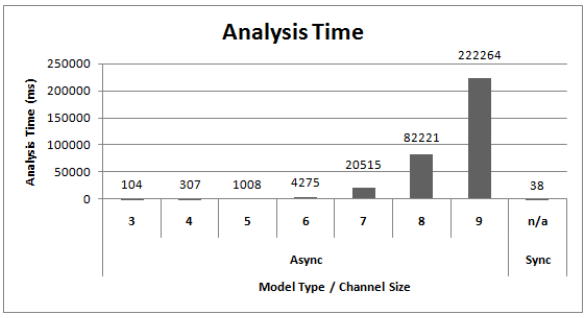


Figure 14: BlowupContract State Machine



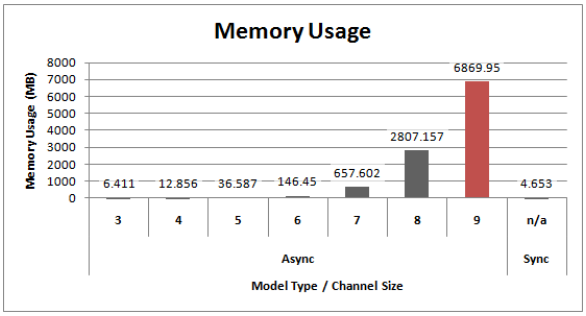


Figure 15: BlowupContract Analysis Time and Memory Usage

the synchronous Promela model for the state machine shown in Figure 14.

As channel size increases, the complexity of the asynchronous analysis grows exponentially. With a channel size of 9, all available memory on our test machine is consumed and analysis fails. However, analysis using the synchronous model takes only 38ms and consumes just over 4.5mb of system memory. Given this potential for exponential blow up, it is possible for realizability analysis in phase 1 to fail and for exhaustive analysis in phase 2 to produce no conclusive results.

However, Singularity enforces a finiteness property on contract state machines [4], which requires that in all cycles at least one message is sent and at least one message is received. This property effectively restricts the channel queue size to a finite bound, and prevents specification of contracts of the type shown in Figure 14. This restriction enables sound analysis. If we pick a channel size that is greater than the length of the longest possible cycle in the contract specification, then, given sufficient memory, the verification results produced using the bounded Promela model are sound. This does not eliminate the potential for exponential state space explosion, however. Figure 16 shows a type of contract which satisfies the finiteness property and is allowed by Singularity's contract verifier, but which will still cause exponential blowup similar to the one shown in Figure 15 when performing exhaustive verification against an asynchronous model. In this case, the complexity of the analysis grows exponentially in the size of the state machine.

Phase 3: LTL Property Verification: In phase 3 of our analysis,

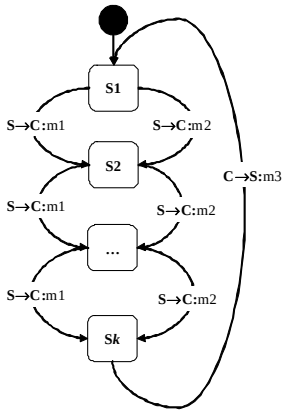


Figure 16: BlowupKContract State Machine

we check that the LTL formulas provided by the user hold for the given channel contract. These LTL formulas specify properties that are expected to hold for any possible message sequence generated by peers that behave according to the contract specification. We use Spin to check these LTL formulas against either the synchronous or asynchronous models. Taking the `KeyboardDeviceContract` from Section 2 as an example, we specified following LTL formulas to characterize expected behaviors of this contract:

```
// Once the Success message is sent, the GetKey or
// PollKey message will eventually be sent
G( Success -> F( GetKey || PollKey ) )
// Once the PollKey message is sent, eventually the
// AckKey or NakKey messages are sent
G( PollKey -> F( AckKey || NakKey ) )
// Once the GetKey message is sent, eventually the
// AckKey or NakKey messages are sent
G( GetKey -> F( AckKey || NakKey ) )
```

If phase 1 was successful, we check the LTL formulas against the more efficient synchronous model. If phase 1 fails, we check the formulas against the full asynchronous model. Table 3 shows a comparison of the efficiency of checking LTL formulas against these two types of models for a representative subset of Singularity contracts.

Analysis Efficiency: Table 4 shows the total processing time of the three phases of our analysis for each of the contracts shown in Table 3. For each of these contracts, phase 2 analysis was sound as we chose a channel size that exceeded the length of the longest cycle (which was 5). In all, we performed phase 1 and phase 2 analyses against 93 contracts in the Singularity code base. Realizability analysis in Phase 1 is very efficient, taking 1.3 ms on average. Exhaustive deadlock detection in phase 2 is considerably more expensive, taking 553ms on average. Among the contracts implemented in version 1.1 of the Singularity RDK, we did not observe the worst case exponential blow-up with respect to channel size in phase 2 of our analysis as described in section 4. However, using a channel size of 8, we observed that the phase 2 analysis time increased with the number of contract states in an exponential trend, as indicated in Figure 17.

These results demonstrate the value of the realizability analysis performed in phase 1. Our analysis found that, all contracts in version 1.1 of the Singularity RDK and all but one in version 2.0 (the `TpmContract` described in Section 2) are realizable, and therefore are deadlock-free and can be verified using synchronous models. This saves the cost of phase 2 analysis and increases the efficiency

Table 3: LTL Verification Performance

Contract Name	Model Type	Transitions	States	Memory (MB)
CompilerPhaseContract	Async	1991	1269	0.338
	Sync	233	168	0.272
DirectoryServiceContract_4	Async	77155	48287	4.643
	Sync	3477	1934	0.254
FatClientContract	Async	2033	1268	0.355
	Sync	175	121	0.282
GameContract	Async	1900	1174	0.353
	Sync	266	179	0.280
GamePlayerContract	Async	1161	758	0.260
	Sync	181	122	0.282
KeyboardDeviceContract	Async	1467	860	0.264
	Sync	147	88	0.284
MapPointProxyContract	Async	1353	800	0.252
	Sync	117	85	0.278
NicDeviceContract	Async	2861	1679	0.340
	Sync	349	209	0.274
SmbClientControllerContract	Async	894	517	0.264
	Sync	126	78	0.283
TcpConnectionContract	Async	20161	14026	1.923
	Sync	555	341	0.252
UdpConnectionContract	Async	15538	8633	1.263
	Sync	1271	601	0.255

Table 4: Processing Time for Each Phase (in ms)

Contract Name	Phase 1	Phase 2	Phase 3 (Sync)	Phase 3 (Async)
CompilerPhaseContract	1	392	38	37
DirectoryServiceContract_4	3	1438	37	219
FatClientContract	1	340	36	49
GameContract	1	370	34	40
GamePlayerContract	1	335	36	40
KeyboardDeviceContract	1	375	36	37
MapPointProxyContract	1	353	46	43
NicDeviceContract	1	401	37	60
SmbClientControllerContract	1	344	37	47
TcpConnectionContract	2	1010	40	82
UdpConnectionContract	1	725	51	68
Average	**1.27**	**553**	**39**	**66**

of the phase 3 analysis. This also indicates that, although Singularity's language semantics and contract verifier allow for unrealizable and even deadlocking contracts, developers, either through intuition or explicit convention, rarely create these types of problematic contracts. Also, the two contracts that failed our realizability test (one from the documentation and one from Singularity RDK version 2.0) allowed interactions that lead to deadlock. Hence our analysis did not generate any false positives in analyzing real contracts although we know that this is possible.

7. RELATED WORK

Our work was inspired by the Singularity operating system. Singularity operating system and its communication mechanisms are presented in [4, 11]. We have not found a discussion of channel contract realizability in the papers on Singularity. The Singularity project focuses on checking conformance of the client and server processes to the channel contracts. However, as our results demonstrate, it is also necessary to analyze the contracts themselves.

Our work builds on the earlier work on realizability of conversation protocols [3, 5–7]. In fact the autonomous condition we use in this paper comes directly from this earlier work. However, the results are significantly different. The conversation protocol model used in [3, 5–7] is a multi-party model (unlike the two party client server model used by Singularity channel contracts). Due to the complexity of the multi-party interactions, the autonomous prop-

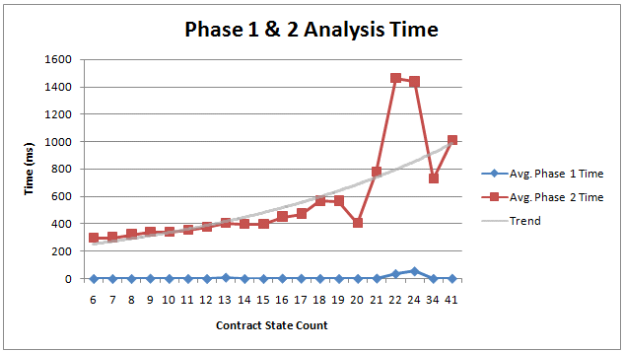


Figure 17: Phase 1 and 2 Analysis Times

erty does not guarantee realizability. There are two other conditions (called lossless join and synchronous compatible in [5, 7]) that are required to guarantee realizability of multi-party protocols. However, checking these extra conditions can be exponential in the worst case, whereas the realizability analysis presented in this paper has linear complexity. To the best of our knowledge, the realizability result reported in this paper (i.e., the autonomous condition is a sufficient condition for the realizability of two-party deterministic conversation protocols) has not been observed before.

The realizability problem dates back to 1980's (see [1, 13, 14]) where it was defined as whether a peer has a strategy to cope with the environment no matter how the environment decides to move. The concept of realizability in this paper is rather different. We are investigating realizability in a closed system and our definition of realizability requires that the implementation generates exactly the same set of behaviors as specified by the contract. The realizability problem for Message Sequence Charts (MSC) [12] or MSC graphs is similar to our definition [2, 16]. However, in the conversation model we use, a global behavior is modeled as a sequence of send events. In many other modeling approaches, e.g., MSCs, both send and receive events are captured. Such different modeling perspectives lead to differences in the expressive power and in the difficulty of analysis and verification problems [7].

Finally, the work on session types [9, 10] also focuses on specification and analysis of interactions among processes. It provides a type theoretic approach where potential communication problems are eliminated by the appropriate restrictions in the type system. The Singularity project is also influenced by the work on session types. However, as we demonstrated in this paper, the type system in Sing# allows specification of contracts that can lead to deadlock.

8. CONCLUSIONS

We showed that Singularity channel contracts can allow deadlocks. We presented a realizability condition that guarantees absence of deadlocks. We built a tool that implements our realizability analysis. We experimented with more than 90 Singularity contracts and identified two contracts with problems. Our experiments demonstrate that efficient analysis of Singularity contracts is feasible. Perhaps, the most surprising outcome of our work is that Sing# programmers were following our realizability condition most of the time although this condition is not stated anywhere in the Singularity documentation. And, in two cases where they failed to follow our realizability condition they specified contracts that allow deadlocks. Finally, Singularity channel contracts are an excellent example of design for verification, where software is structured in ways that enable effective verification. Without specification of channel contracts the problems we identified would be buried in the code and would be hard to extract and analyze.

9. REFERENCES

[1] M. Abadi, L. Lamport, and P. Wolper. Realizable and unrealizable specifications of reactive systems. In *Proc. of 16th Int. Colloq. on Automata, Languages and Programming*, volume 372 of *LNCS*, pages 1–17. Springer Verlag, 1989.

[2] R. Alur, K. Etessami, and M. Yannakakis. Realizability and verification of MSC graphs. In *Proc. 28th Int. Colloq. on Automata, Languages, and Programming*, pages 797–808, 2001.

[3] T. Bultan, X. Fu, R. Hull, and J. Su. Conversation specification: A new approach to design and analysis of e-service composition. In *Proc. 12th Int. World Wide Web Conf.*, pages 403–410, May 2003.

[4] M. Fähndrich, M. Aiken, C. Hawblitzel, O. Hodson, G. C. Hunt, J. R. Larus, and S. Levi. Language support for fast and reliable message-based communication in singularity os. In *Proc. 2006 EuroSys Conf.*, pages 177–190, 2006.

[5] X. Fu, T. Bultan, and J. Su. Conversation protocols: A formalism for specification and analysis of reactive electronic services. *Theoretical Computer Science*, 328(1-2):19–37, November 2004.

[6] X. Fu, T. Bultan, and J. Su. WSAT: A tool for formal analysis of web services. In *Proc. 16th Int. Conf. on Computer Aided Verification*, pages 510–514, 2004.

[7] X. Fu, T. Bultan, and J. Su. Synchronizability of conversations among web services. *IEEE Transactions on Software Engineering*, 31(12):1042–1055, December 2005.

[8] G. J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley, Boston, Massachusetts, 2003.

[9] K. Honda, V. T. Vasconcelos, and M. Kubo. Language primitives and type discipline for structured communication-based programming. In *7th European Symposium on Programming on Programming Languages and Systems (ESOP'98)*, pages 122–138, 1998.

[10] K. Honda, N. Yoshida, and M. Carbone. Multiparty asynchronous session types. In *Proc. 35th Symp. on Principles of Programming Languages*, pages 273–284, 2008.

[11] G. C. Hunt and J. R. Larus. Singularity: rethinking the software stack. *Operating Systems Review*, 41(2):37–49, 2007.

[12] Message Sequence Chart (MSC). ITU-T, Geneva Recommendation Z.120, 1994.

[13] A. Pnueli and R. Rosner. On the synthesis of a reactive module. In *Proc. 16th ACM Symp. Principles of Programming Languages*, pages 179–190, 1989.

[14] A. Pnueli and R. Rosner. On the synthesis of an asynchronous reactive module. In *Proc. 16th Int. Colloq. on Automata, Languages, and Programs*, volume 372 of *LNCS*, pages 652–671, 1989.

[15] Singularity design note 5 : Channel contracts. singularity rdk documentation (v1.1). `http://www.codeplex.com/singularity`, 2004.

[16] S. Uchitel, J. Kramer, and J. Magee. Incremental elaboration of scenario-based specifications and behavior models using implied scenarios. *ACM Transactions on Software Engineering and Methodology*, 13(1):37–85, 2004.