Verification Techniques To Avoid Deadlocks On Channel Contracts

Shahzina Mubeen
Department of Computer Science, University of Lahore, Sargodha Campus, Pakistan

&

Madam Nida Yasir
Department of Computer Science, University of Lahore, Sargodha Campus, Pakistan

ABSTRACT-
Microsoft singularity is a research operating system. Singularity is a highly dependable OS. Its basic architecture features are SIPs, the contract-based channels and the manifest-based programs. Our work is motivated by the Singularity operating system. The claim of singularity guarantee that there is no deadlock on channel contracts, This claim is wrong. We present channel contract analysis and many techniques. By using these techniques and tool efficient analysis of singularity channel contract is possible.

1- INTRODUCTION

Inter process communication is used to communicate between different process and services within the system. Shared memory and message passing is fundamental model of IPC that are commonly used. Singularity is an experimental operating system developed by Microsoft research. Message passing model of IPC with contract-based communication channels are used in this operating system. In singularity processes are not allowed to share memory with each other or the kernel. Communication through Singularity channels corresponds to asynchronous communication via FIFO queues. When a process sends a message through a channel, the message is appended to a message queue.[1] Channels are bi-directional and the only means by which processes in Singularity can communicate. The channel is typed by contract which defines exactly what messages can be sent along the channel, as well as the sequence in which the messages must be sent. If the contract is violated, as isn't caught at compile time, the channel is closed.

When data is sent between two processes across a channel, the ownership of that data is transferred from the sender to the receiver. Since only one process can manage a piece of data at a time the risk of such data being used by other processes is greatly reduced. Communication channels are governed by contracts that are statically defined, meaning they cannot be changed after installation. The main properties enforced by channel contracts are 1) senders send messages that are expected by receivers, 2) receivers are capable and willing to handle all messages allowed by the contract, and 3) senders and receivers that have been verified separately against a given contract cannot deadlock with communicating over a channel governed by such contract.

In Singularity, each channel is governed by a channel contract [3]. 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.

A. Singularity Channels
Channels allow 2-Party asynchronous communication via FIFO message queues. Asynchronous means 1) Sends are non-blocking, 2) Receives block until a message is at the head of a receive queue. Data is exchanged over bidirectional channels, where each channel consists of exactly two endpoints (called Imp and Exp). At any point in time, each channel endpoint is owned by a single thread [3]. Buffers and other memory data structures can be transferred by pointer, rather than by copying. These transfers pass ownership of blocks of memory. Buffers do not permit sharing between the sender and receiver since static verification prevents a sender from accessing memory it no longer owns.
B. 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 defueling and processing messages sent to an endpoint [3].

a. Contracts specify two things:
   1. The messages that may be sent over a channel
      • out message are sent from the Server endpoint to the Client endpoint (SàC)
      • in messages are sent from the Client endpoint to the Server endpoint (CàS)
   2. The set of allowed message sequences
      • out message marked with ! in messages marked with ?

b. Example of keyboard device contract.
   public contract KeyboardDeviceContract
   {
      state Start: {
         Success! -> Ready;
      }
      state Ready:
      {
         GetKey? -> Waiting;
         PollKey? -> (AckKey! or NakKey!) -> Ready;
      }
      state Waiting: {
         AckKey! -> Ready;
         NakKey! -> Ready;
      }
   }

C. Problem Statement
The claim of singularity of paper, 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". This claim is wrong. We present channel contract analysis in this paper uncovered two contracts which show this statement to be false. Two Singularity channel contracts that allow deadlocks, one from the Singularity documentation and one from the Singularity code distribution. These two contracts are shown below

a) Reservation Session Contract:
The first Singularity contract we discovered demonstrating the potential for deadlock is called the Reservation Session contract. It is an example contract given in the Singularity RDK documentation [7]. It defines the state machine shown in Figure 1. Table 1 shows an interleaving of valid client and server actions according to this contract that leads to a deadlock.

<table>
<thead>
<tr>
<th>Server Action</th>
<th>Time Step</th>
<th>Client Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Send: Request</td>
<td>T0</td>
<td></td>
</tr>
<tr>
<td>Send: Cancel</td>
<td>T4</td>
<td></td>
</tr>
</tbody>
</table>

Table 1: Reservation Session Deadlock Scenario

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, nor is the channel deadlocked. Although this contract is given in the Singularity documentation, it is not
included as code in the Singularity distribution. Singularity processes can deadlock even when they are faithfully following a channel contract.

**b) TpmContract:**
The second contract violating the claim that Singularity contracts are deadlock-free, is the TpmContract. TpmContract is intended to be used to communicate with the “Trusted Platform Module”, which is a cryptography processor used to authenticate hardware and software running on the platform.

This contract is included in version 2.0 of the Singularity RDK and is used by a component of the Singularity kernel. 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 2. 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 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 trans- sitions 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. The major contributions of this paper are: We have discussed many types of verification techniques of channel contracts and which technique is best and give guarantee of no deadlock is well explained in this review. KeyboardDeviceContract is realizable whereas ReservationSession and TpmContract are not. Moreover, we give a sufficient condition for the realizability of channel contracts.
2- BACKGROUND LITERATURE REVIEW

State machine construction allows for automated verification and analysis of channel communication. Singularity compiler automatically checks compliance of client and server processes to the specified contract. Singularity uses a combination of run time monitoring and static verification. It is not very difficult to figure out that the channel contracts can allow deadlock but provable: two processes that try to exchange two messages, on two different channels, by first waiting for the message of the other process before sending its own message. The problem here resides in the fact that the two channels are ruled by different contracts, and the deadlock can only be ruled out if there is a global discipline over all channels, and more generally over all synchronization primitives.

Existing literature provides several methods for avoiding deadlocks: Kobayashi’s type system is one example that applies to the synchronous \( \pi \)-calculus [14]. Based on global session descriptions, Bettini et al. developed a framework where it is possible to establish global progress for multi-channel protocols. More recently, Leino et al. introduced a program logic that ensures deadlock-freedom for programs that manipulate channels and locks.

Finally, the work on session types [15] 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 influence 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. Following techniques are used for verification of contracts. First two techniques are used in singularity.

a. Model checking
   Runtime monitoring drives a contracts’ state machine and watches for erroneous transitions [1].

b. Static Analysis based on conformance checking
   Static checker verifies safety properties. Statically ensure deadlock avoidance. Contracts are verified with a more general static analysis based on conformance checking [13].

c. Realizability Analysis
   if the processes whose communication behavior in terms of conversations (i.e., send sequences) is equal to the set of conversations (i.e., send sequences) specified by the conversation protocol then they are realizable.

3-METHODOLOGY
   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. It is also necessary to analyze the contracts themselves.

a) Conversation Protocol
   Conversation protocols identify the global communication behavior. How do we implement processes that conform to the conversation protocol?
   Realizability question:
   – *Given a conversation protocol, are there processes whose communication behavior in terms of conversations (i.e., send sequences) is equal to the set of conversations (i.e., send sequences) specified by the conversation protocol?*
   – So we need two requirements for realizability:
   – Conversations specified by the conversation protocol = Conversations generated by the asynchronous system
   – Asynchronous system is well-formed: *All sent messages can be eventually consumed*
   – Conversation protocol is realizable if and only if there exists such an asynchronous system

b. Conditions for realizability (no message content)
   • *Lossless join*
   – Conversation set should be equivalent to the join of its projections to each peer
   • *Synchronous compatible*
   – When the projections are composed synchronously, there should not be a state where a peer is ready to send a message while the corresponding receiver is not ready to receive
   • *Autonomous*
   – At any state, each peer should be able to do only one of the following: send, receive or terminate.
   – **Connection between realizability and synchronizability:**
   – A conversation protocol is realizable if its projections to peers are synchronizable and the protocol itself satisfies the lossless join condition

   c. Realizability Problem
   Following contracts in singularity are not realizable.
   • KeyboardDeviceContract is not realizable
   – It violates the autonomous condition and it turns out that autonomous condition is sufficient (but not necessary) for realizability of two-party protocols (Singularity channel contracts are two-party protocols). If a contract is autonomous, it is guaranteed to be realizable. However, it can be realizable but not autonomous. i.e., false positives are possible when we use autonomous condition as our realizability check
• TpmContract is not realizable

It violates the autonomous condition. As I mentioned earlier, autonomous condition is sufficient (but not necessary) for realizability of two-party protocols (Singularity channel contracts are two-party protocols). If a contract is autonomous, it is guaranteed to be realizable. However, it can be realizable but not autonomous. i.e., false positives are possible when we use autonomous condition as our realizability check.

d. Difference

• Key difference between the KeyboardDeviceContract and the TpmContract. Or to put it another way, there is no sequence of messages that can be sent that would get either process into a state where they are stuck or are unable to finish.
• More formally, we say the KeyboardDeviceContract is realizable whereas the TpmContract is not realizable.
• Which brings me to the Realizability problem, which is the central problem this work is addressing
  Simply stated, the realizability problem is this…
  • In the paper we formally define “well-behaved”, but essentially what it means is that both processes are always able to reach a configuration in which they are in a final state and their receive queues are empty or they’ve been able to consume all the messages that have been sent to them.
  • If a contract is realizable, it’s guaranteed that there are implementations of that contract that conform to it and fully express it. And that are guaranteed not to deadlock. This is essentially the guarantee that the Singularity documentation was claiming their conformance checks provided. This notion of realizability begs the question: How do you check if a contract is realizable, and is that check efficient or even possible?

 e. CADP Tool Box

• Implemented using CADP toolbox for checking synchronizability of Singularity Channel Contracts.
  We used the front end of an earlier tool called Tune for analyzing Singularity channel contracts [Stengel, Bultan ISSTA 2009]
  • which can then be used for model checking behaviors of synchronizable channel contracts.
  • Then we use the equivalence checking algorithms implemented in CADP toolbox to check their equivalence. Using Tune, we can also generate Promela specifications for synchronized version
  Checked synchronizability of 86 Singularity Channel Contracts.
  – Synchronous Systems:
    2 to 23 states; 1 to 60 transitions.
  – Asynchronous System with buffer size 1: 3 to 99 states; 2 to 136 transitions
  • 84 Contracts are synchronizable. 2 contracts that are not synchronizable cause deadlocks!
  – i.e., they are buggy!

F. Tools for analyzing channel contracts

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

A tool called Tune for analyzing Singularity channel contracts written in Sing#[12]. 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 [7], with the following limitations:
1. Tune requires all messages specify explicit directional qualifiers. Messages must be sent either from server to client, or from client to server [9].

In Sing# contract specification message parameters do not influence the contract behavior, so we are able to ignore them during our analysis. 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.

1) 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 [9].

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 non-deterministically 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.
In tune model there are three phases 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.

2) Synchronous Promela Model:
The full explicit state verification performed against asynchronous contract models is, in general, unstable 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 deadlockfree 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.

b. Reliability Analysis and Solution:
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 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 positive. If it finds a deadlock, a counterexample 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. 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. How-ever, analysis using the synchronous model takes only 38ms and consumes just over 4.5mb of system memory. Figure 6 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 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.

c. Autonomous condition and false positives
• Example: TpmContract
  Since autonomous condition is not a necessary condition, it can cause false positives when used for checking realizability.
  • Using our recent results we can show that this modified protocol is realizable using the necessary and sufficient condition for realizability. Here’s the TpmContract that I showed before was not realizable because it is not well behaved or can lead to deadlock. We can see that it also violates the autonomous condition here in the IO_RUNNING state. Now, this contract can be fixed and made realizable by adding one state and two transitions, like so.
  • This contract is now realizable and won’t lead to deadlock, but it still violates the autonomous condition. So, we wanted to find a way to resolve these types of false positive, so that we could know with more certainty if a contract is fact unrealizable if the autonomous check fails.
Fig. 5

This leads to another significant benefit of realizability analysis. If a contract is realizable, the conversations generated. This means we can use a synchronous communication model for verification. The cost of the same analysis against a synchronous model grows linearly in the # of these states, which on the scale of this graph is basically a flat line.

4- CONCLUSION AND FUTURE WORK

Our work was motivated by the Singularity operating system. Singularity claims that there is no deadlock between client and server processes that are verified with respect to the channel contract. This claim is wrong. We present channel contract analysis in this paper uncovered two contracts which show this statement to be false.

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 shows that, it is also necessary to analyze the contracts...
themselves. We showed that Singularity channel contracts can allow deadlocks. We presented a realizability condition that guarantees absence of deadlocks. We use a tool that implements our realizability analysis. By using this tool our experiments demonstrate that efficient analysis of Singularity contracts is feasible. Finally, Singularity channel contracts are an excellent example of design for verification, where software is structured in ways that enable effective verification. In this review we have discussed many verification techniques and tools to verify channel contracts to ensure deadlock avoidance.

5. REFERENCES
