Improving Live Sequence Chart to Automata Transformation for Verification

Rahul Kumar[1] **and Eric G Mercer**[2]

[1] rahul@cs.byu.edu, [2] egm@cs.byu.edu
Computer Science Department
Brigham Young University
Provo, UT, USA

Abstract:

This paper presents a Live Sequence Chart (LSC) to automata transformation algorithm that enables the verification of communication protocol implementations. Using this LSC to automata transformation a communication protocol implementation can be verified using a single verification run as opposed to previous techniques that rely on a three stage verification approach. The novelty and simplicity of the transformation algorithm lies in its placement of accept states in the automata generated from the LSC. We present in detail an example of the transformation as well as the transformation algorithm. Further, we present a detailed analysis and an empirical study comparing the verification strategy to earlier work to show the benefits of the improved transformation algorithm.

Keywords: live sequence charts, transformation, automata, verification, non-determinism

1 Introduction

Current trends in system development are shifting towards creating and developing larger systems using several smaller communicating sub-systems. With the increasing popularity of such modular designs comes the burden of creating, implementing, and testing the implemented communication protocols. Specification of communication protocols has been explored significantly in the past. English, which has been traditionally used as the most common language for specifying protocols, lacks the formal rigor and preciseness needed for clarity. Viable alternatives are formal specification languages such as UML, Message Sequence Charts (MSCs) and Live Sequence Charts (LSCs) [IT93, DH99, BDK+04]. The evolution of these graphical languages has led to their application to modeling and specifying communication behaviors in a variety of different domains [BHK03, KHG05, DK01]. Other research has also investigated the automatic synthesis of systems from LSCs as well as the verification and validation of requirements on the LSCs themselves [HK01, AY99, SD05]. Efficient methodologies for using these graphical languages in a formal verification environment provide the support in the development process to completely certify, test and develop a system. Since LSCs are a more expressive and semantically rich visual specification language compared to MSCs, Timing Diagrams and Sequence Diagrams in UML, we focus on techniques related to LSCs. Due to the encompassing nature of LSCs, the techniques and algorithms presented in this paper are also applicable to the afore mentioned specification languages.

Previous work in [KTWW06, Klo03] presents a strategy to verify systems against LSC specifications by transforming the LSC to a *positive* automaton. We use the term positive automaton to denote automaton that witness chart completions. With the positive automaton, a system is verified against the LSC in three stages: reachability analysis for detecting safety violations, ACTL verification for detecting liveness errors, and finally, if the first two steps fail to provide a significant result, full LTL verification is required to completely verify the system. The authors argue that the verification algorithms are applied in increasing order of cost and for certain sub-classes of LSCs not all algorithms need to be applied, which can eventually save on the total verification cost. Although the approach presented in [KTWW06] is sound, it has several drawbacks. For any arbitrary LSC, the approach at a minimum has to apply reachability analysis as well as ACTL model checking for verifying the safety and liveness properties of the system against the LSC. In the worst case, LTL verification is required to completely verify the system, which was shown to be impractical for LSC verification [KM07]. Another drawback of the verification approach is the specialized algorithms and tools that have to be created to perform the verification, which limit the general applicability and acceptance of the approach. The approach presented in this paper only requires one verification algorithm of the same cost as reachability analysis to completely verify a system against any arbitrary LSC.

We present a direct and obvious transformation of the LSC to a *negative* automaton by changing the placement of accept states. We use the term negative automaton to denote automaton that witness chart violations as opposed to chart completions. Using this improved LSC to automaton transformation a system can be formally verified against the LSC specification by performing only language containment on the parallel composition of the system automaton and the negative automaton of the LSC created using the transformation algorithm presented in this paper. Additionally, this approach does not require the use of customized algorithms and tools to verify a system against a specification. Using our new LSC to automaton transformation, we verify systems against larger more concurrent LSCs that were previously not verifiable with direct LSC to LTL or LSC to positive automaton transformations.

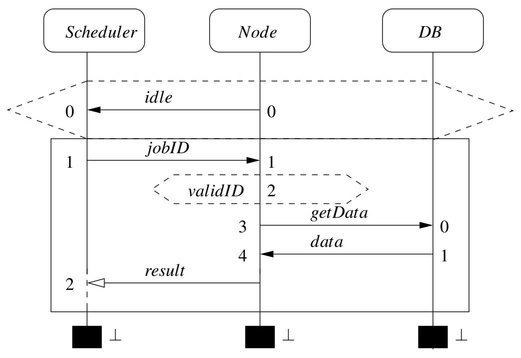
The structure of the paper is as follows. Section 2 presents a brief introduction to LSCs and an overview of the basic LSC to automaton transformation algorithm as described in [Klo03]. Section 3 discusses in detail an example of using our approach for verifying a system against an LSC. This example will be used for the remainder of the paper as well. Section 4 discusses the details of the transformation algorithm and presents the theoretical results to prove the correctness of the transformation algorithm. Section 5 presents an analysis of the improved transformation compared to the old transformation presented in [Klo03]. Section 6 presents a subset of the results using the improved verification approach in both symbolic and explicit state model checkers. Finally, 7 discusses the conclusions and future work. Proofs, details and additional results can be found in the long version of the paper at `http://vv.cs.byu.edu/~rahul/lsc2automata.pdf`.

2 LSC Overview

We briefly introduce some constructs of the LSC grammar[1]. Fig. 1(a) shows an example LSC where an idle node in a compute cluster requests and processes a job from the scheduler's queue with a possible implementation of the *Node* and *DB* process in Fig. 1(b). There are three *processes* in the example LSC: *Scheduler*, *Node* and *DB*. Each process is drawn with a rectangular instance head and a vertical life-line originating from each instance head. The life-line represents the time dimension in the LSC with time progressing in the downward direction. Communication between processes occurs via *messages* with the arrows representing the direction of communication. The *idle* message is an example of a *synchronous* message (filled arrowhead) where both the sender and receiver have to be ready for the message to be observed. The actual message communication occurs instantaneously for the sender and receiver. The *result* message is an example of an *asynchronous* message (unfilled arrowhead) where the sender does not have to block for the receiver to be ready to receive the message. The send event is written as *result*! and the receive event is written as *result*?. The example LSC also contains a *cold non-bonded condition* (second dashed hexagon) which enforces the *validID* predicate after a *jobID* has been received from the *Scheduler*. If the condition is violated, the *Node* process exits the chart. On each life-line any point where a condition or an event occurs is referred to as a *location*. Locations are unique to each life-line and in our research are represented by numbers next to the instance life-line. By default all locations are *hot* or *mandatory* locations unless specified otherwise using a dashed line for the life-line. The location for receiving the *result* message in the *Scheduler* life-line is the only cold location in the example chart. The behavior specified on a cold location is not mandatory, which implies that the *result* message may or may not be received by the *Scheduler*. Finally, behaviors described by the LSC are partitioned into the *pre-chart* (dashed hexagon before solid rectangle) and the *main chart* (rectangle after pre-chart). The pre-chart specifies the activation condition of the LSC and the main chart describes the behavior which must follow the pre-chart. In the example LSC, the main chart is a *universal* main chart (solid line), which represents behaviors that have to be observed every time the pre-chart is satisfied.

In addition to the constructs shown in the example LSC, several other constructs are also available. The main chart can be specified as an *existential* chart (drawn with a dashed rectangle) that specifies behavior the system must satisfy at least once when the pre-chart is satisfied (as opposed to every time the pre-chart is satisfied). Conditions if attached to another event are *bonded* otherwise *non-bonded*. By attaching conditions to other events, the condition is evaluated at the exact moment the bonded event occurs, as opposed to non-bonded conditions where the condition is continuously evaluated until satisfied. LSCs also allow the specification of *invariants* which are conditions spanning over multiple events in the LSC. *Co-regions* specified with a dashed line parallel to a life-line allow events to occur in any order. For example, if the messages *getData* and *data* are specified in a co-region, either message *data* or *getData* may occur first. It is only necessary for all events in a co-region to occur. Finally, conditions, messages, and locations may be specified as *hot* or *cold*. If drawn with a solid line, the construct is hot and specifies mandatory behavior, and if drawn with a dashed line, the construct specifies cold or provisional behavior.

[1] See [DH99, BDK^{+}04] for details.



(a)

```
Process Node:
  if(idle) then
    Send(``idle'', Scheduler)
    Receive(``jobID'', Scheduler)
    if(not ``validID'')
      break
    Send(``getData'', DB)
    Receive(``data'', DB)
    Send(``result'', Scheduler)
  endif
End Process Node
Process DB:
  while(true){
    Receive(msg, Node)
    if(msg is ``data'')
      Receive(``data'', Node)
    endif
      RemoveData(data)
    endif
  end while
End Process DB
```

(b)

Figure 1: An example specification describing the interaction between a cluster node (*Node*), a database (*DB*) and a job scheduler (*Scheduler*), and a possible implementation of the *Node* and *DB* processes (a) The example LSC containing a subset of the complete LSC grammar (b) A system implementing the *Node* and *DB* processes described in the LSC.

Our method supports all the mentioned constructs of LSCs with the following commonly accepted restrictions. First, we adopt the *strict* interpretation of LSCs (i.e., no duplicate message instances are allowed within a chart). Second, the LSC and all charts within the LSC are to be acyclic. Third, we also do not consider overlapping LSCs or iterative LSCs (Kleene stars) where multiple instances of the chart may be executed simultaneously. Since most scenario based specifications in general do not deal with the constructs omitted from this research, the restrictions do not affect the general applicability of our results.

2.1 Transforming Live Sequence Charts to Automata

Past research in the area of transforming LSCs to automaton has primarily revolved around the generation of positive automaton that detect chart completions [Klo03, HK01, BH02, KW01]. Work in [Klo03] gives a detailed presentation of the algorithm to transform an LSC to positive automaton. We present an overview of this algorithm followed by a discussion of some key aspects of the algorithm.

The LSC to automaton unwinding algorithm explores all possible inter-leavings of the events defined in the LSC starting from the top and ending at the bottom of each life-line in the chart. The possible event inter-leavings are explored by considering the partial order induced by the semantics of the LSC. The partial order of the chart dictates that the locations in each instance are totally ordered unless part of a co-region; thus, implying that each instance has to progress linearly from top to bottom. For example, in the chart shown in Fig. 1(a), instance *Node* cannot move from location 1 to location 4. From location 1, *Node* has to move to the next logical location: location 2. To maintain the current state of the LSC, we define a *cut* as a set of locations in the chart with exactly one location for each instance. The cut is used to record the current state of the chart and create successor cuts. The reachable set of cuts from the initial cut is the automaton for the chart. Each state of the automaton corresponds to a reachable cut of the chart. Successor cuts are generated using the set of enabled transitions for a given cut. The initial cut for all charts is created by placing each instance at its first location, $(0,0,0)$, where the first, second and third locations correspond to the locations for the *Scheduler*, *Node* and *DB* instances.

The enabled set of transitions for a cut is created using the chart semantics. For example, a synchronous message is enabled if both the sender and receiver of the message are at their respective send and receive locations. In our chart, the message *idle* is observed if the *Scheduler* and *Node* instances are each at locations 0. At the initial cut, $(0,0,0)$, the *idle* message is enabled. On the other hand, since the *Node* is not at location 3, the *getData* message is not enabled in the initial cut, even though the *DB* is at location 0. When the *idle* message is explored from the enabled set, a successor cut is generated where the locations for the involved instances have been updated. In this case, the locations for the *Node* and *Scheduler* instances are updated to their next logical location giving us the successor cut $(1,1,0)$. At the cut $(1,1,0)$, the *jobID* message is enabled, which leads to the cut $(2,2,0)$. Asynchronous sends are enabled by default when the corresponding instance is at the send location and asynchronous receives are enabled only if the corresponding send event has occurred and the receiving instance is at the receive location. Conditions act as a synchronization point where each participating instance should be at its respective condition location for the condition to be evaluated. A full description of these semantics can be found in [Klo03]. Multiple enabled transitions lead to multiple successor cuts from the given cut representing the concurrency in the chart.

Using the chart semantics, successor cuts are generated from the initial cut and each unique cut is processed until the final cut is reached where each instance is at the bottom of its life-line. Each unique cut of the chart corresponds to a state in the final automaton. The initial cut $(0,0,0)$ corresponds to state q_0 in Fig. 2(a). The successor cut $(1,1,0)$ corresponds to the state q_1 where the *idle* message has already been observed and the next message to be observed is *jobID*. Cut $(2,2,0)$ corresponds to state q_2 and the final cut corresponds to state q_7 where no further events are to be observed. Notice that transitions taken to generate successor cuts correspond to the transition labels in the automaton.

Finally, to create the positive automaton from the LSC, states corresponding to legal exits of the chart are marked as accept states. For example, state q_7 in Fig. 2(a) is marked as an accept state because it corresponds to the final cut of the LSC which represents a legal completion of the chart. Additionally, state q_6 is also marked as an accept state since it corresponds to the cut where the cold message *result* does not have to be received.

From the automaton in Fig. 2(a) we also notice that state q_2, where cold condition *validID* occurs is non-deterministic. This non-determinism is a result of the adopted semantics of cold conditions in [Klo03]. If *validID* is not satisfied, the automaton can either stay in state q_2 and wait for the condition to be satisfied or move to the exit state q_{exit} to signify that the cold condition was not satisfied and the chart has exited successfully. This non-determinism resulting from non-bonded conditions forces the approach of [Klo03, KTWW06] to translate the LSC automaton to an LTL property and re-perform the verification using the LTL property, which has been shown to be ineffective for even moderate size charts due to the size of the resulting LTL formula [KM07].

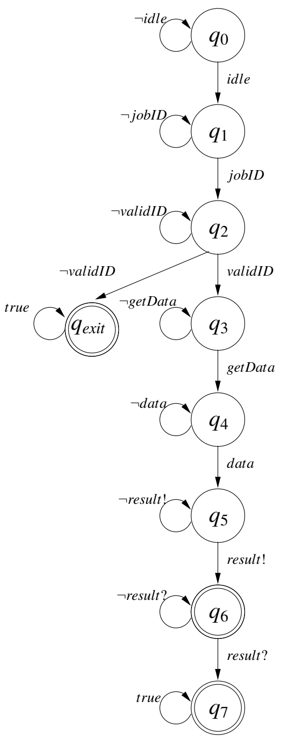
3 Transformation and Verification Example

We use the automaton produced by the unwinding algorithm discussed earlier as our initial automaton. The initial automaton from the unwinding algorithm is shown in Fig. 2(a). We transform this positive automaton to a negative automaton that can be used in our single pass verification approach. Fig. 2(b) and (c) show the transformed negative automaton.

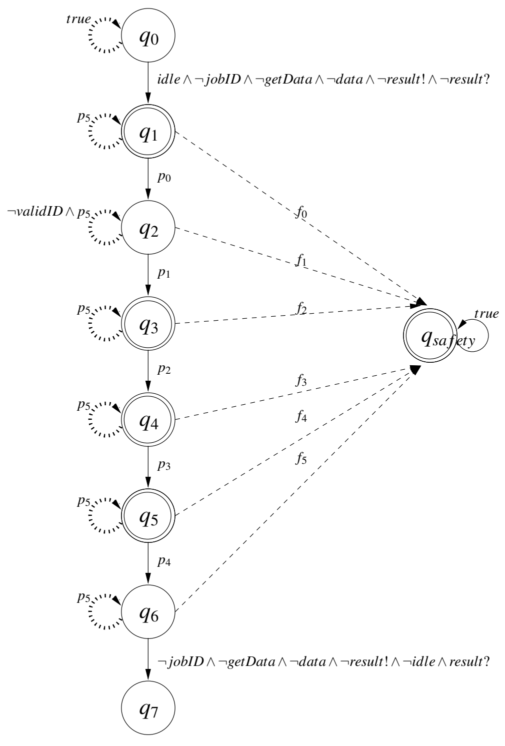
Our approach transforms the LSC chart to a negative automaton capable of detecting chart violations (as opposed to chart completions) that is naturally suited for verifying systems using language containment. The first step in the transformation process is to remove all the accept labels from the automaton. Next the exit state q_{exit} and any transitions leading to the exit state are removed from the initial automaton. In our example of Fig. 2(a) we remove the transition from state q_2 to state q_{exit}, which also removes the non-determinism from the automaton arising from the non-bonded condition. The algorithm then introduces safety transitions (dashed edges in Fig. 2(b)) from all states that contain a transition belonging to the main chart to the safety state q_{safety}. The safety state is an accept state introduced in the automaton to capture all safety violations in the system. It has a single outgoing transition to itself predicated on $true$. The safety transitions enable the detection of safety violations which consist of duplicate messages (messages that have been observed before) and out of order messages in states that correspond to main chart states. For example, in state q_1 of Fig. 2(b), the only legal transition is if the $jobID$ message is observed. Since $jobID$ is a main chart transition, state q_1 corresponds to a main chart state and a safety transition is introduced. The safety transition $idle \vee getData \vee data \vee result! \vee result?$ from state q_1 to q_{safety} is taken if any message except $jobID$ is observed.

After the introduction of safety transitions, the algorithm updates the self-loops on each state (dotted edges in Fig. 2(b)). The self-loops enable the automaton to remain in a given state until an event forcing progress is observed. For example, in the automaton shown in Fig. 2(b), state q_4 has a self-loop, $\neg idle \wedge \neg jobID \wedge \neg getData \wedge \neg data \wedge \neg result! \wedge \neg result?$, that is taken until the $data$ message is observed, which moves the automaton to the next state q_5. The only exception is the self-loop for the first state and the final state. The first state q_0 contains a self-loop with the $true$ annotation to capture all possible future instances (and possible errors) of the chart in a reactive system. The final state does not have any self-loops. This is because the final state represents the successful completion of the chart and no further errors are possible unless a new chart instance is observed, which is detected in the first state.

Finally, the algorithm marks illegal end points of the main chart as accept states to facilitate detection of chart violations. For example, state q_1 in Fig. 2(b) is at the beginning of the main chart where the message $jobID$ is yet to be received. If the $jobID$ message is never observed, the automaton remains in state q_1 indefinitely, which should be reported as an error. To report this error, state q_1 is marked as an accept state. States containing no transitions corresponding to hot constructs in the main chart are not marked as accept states. For example, in Fig. 2(b), state q_2 is not marked as an accept state because the $validID$ condition is a cold condition, and its absence does not result in an error. State q_0 is not marked as an accept state either because it does not contain any outgoing transitions corresponding to a hot construct in the main chart. If



(a)



(b)

f_0	$idle \lor getData \lor data \lor result! \lor result?$	f_1	$idle \lor jobID \lor getData \lor data \lor result! \lor result?$
f_2	$idle \lor jobID \lor data \lor result! \lor result?$	f_3	$idle \lor jobID \lor getData \lor result! \lor result?$
f_4	$idle \lor jobID \lor getData \lor data \lor result?$	f_5	$idle \lor jobID \lor getData \lor data \lor result!$
p_0	$jobID \land \neg idle \land \neg getData \land \neg data \land \neg result! \land \neg result?$	p_1	$validID \land \neg jobID \land \neg idle \land \neg getData \land \neg data \land \neg result! \land \neg result?$
p_2	$getData \land \neg idle \land \neg jobID \land \neg data \land \neg result! \land \neg result?$	p_3	$data \land \neg idle \land \neg jobID \land \neg getData \land \neg result! \land \neg result?$
p_4	$result! \land \neg idle \land \neg jobID \land \neg data \land \neg getData \land \neg result?$	p_5	$\neg idle \land \neg jobID \land \neg getData \land \neg data \land \neg result! \land \neg result?$

(c)

Figure 2: The initial and transformed automaton for the example LSC shown in Fig. 1(a). (a) the initial automaton (b) the transformed automaton and (c) list of transition labels.

the *idle* message is never observed, the pre-chart is not satisfied, which is not a violation of the specification. State q_6 is not marked as an accept state since the location of the *result*? event is cold implying that the *result*? event does not have to be observed. Finally, state q_7 in Fig. 2(b) is also not marked as an accept state since it is the final state where the behavior as described in the universal chart has been satisfied without errors.

Verification of the system is performed by first creating the system automaton in the usual manner. We verify the parallel composition of the system automaton and the negative automaton of the LSC by searching the behavior space of the intersection for accepting cycles. Any cycles detected correspond to errors in the system. Fig. 1(b) shows a possible implementation of the *Node* and *DB* processes in a cluster. The *Scheduler* process has not been shown in the implementation but is assumed to be correctly implemented. When idle, the *Node* process requests a job from the scheduler (line 2). The *Node* process then waits to receive the *jobID* and validates the *jobID* using the predicate *validID* (lines 3 - 5). Next, the *Node* process requests data from the *DB* (line 6), processes the data and sends the result to the *Scheduler* (lines 7 - 8). The *DB* process receives and processes messages as they arrive (lines 12 - 19). In this particular implementation, the *DB* process is erroneous because it never receives/processes the *getData* message from the *Node*. Since the *getData* message is a synchronous message and the *DB* process is never ready to receive the *getData* message, the *Node* and *DB* processes never progress even though they should. Verification of the parallel composition of the system automaton (not shown) with the property automaton in Fig. 2(b) produces the word $(idle, jobID, validID, (\neg getData)^{\omega})$, with the corresponding trace: $(q_0, q_1, q_2, (q_3)^{\omega})$, where ω indicates infinite repetition. Since q_3 is marked as an accept state, the trace is reported as an accepting cycle and the violation has been discovered. Using the positive automaton in the verification approach of [KTWW06] requires two verification runs of comparable complexity to detect the same violation.

4 Transformation and Verification Details

The transformation presented in this work is based on language containment and automata theory. We use *Symbolic automata*, an extension of Büchi automata, that allows observing any of a possible set of inputs on an edge. Formally Symbolic automata are given by $A = \langle \Sigma, Q, \Delta, q^0, F \rangle$ where, Σ is the finite *alphabet* of input symbols (variables), Q is the finite set of states, $q^0 \in Q$ is the initial state, $F \subseteq Q$ is the set of final/accepting states, and $\Delta \subseteq Q \times \phi \times Q$ is the transition relation. A transition $(q, \phi, q') \in \Delta$ represents the change from state q to state q' when the formula ϕ is satisfied.

We partition the set of Boolean variables Σ into three distinct sets Σ_{msgs}, $\Sigma_{\text{invariants}}$, and $\Sigma_{\text{conditions}}$, that contain the Boolean variables that are used for messages, invariants and conditions in the chart respectively. For the chart shown in Fig. 1(a), $\Sigma_{\text{msgs}} = \{idle,\ jobID,\ getData,\ data,\ result?,\ result!\}$ and $\Sigma_{\text{conditions}} = \{validID\}$. The set $\Sigma_{\text{main}} = \{jobID, validID, getData, data,\ result?,\ result!\}$ is the set of Boolean variables that are used in the main chart only. We also have a set $\Delta_{\text{hot}} \subseteq \Delta$ which only contains transitions that correspond to hot constructs in the chart (hot messages, hot conditions etc.).

For a set of Boolean functions $\Gamma = \{\phi_0, \phi_1, ..., \phi_n\}$ we define the function $disjunct(\Gamma)$ which returns the disjunct of the individual formulas in Γ and the function $conjunct(\Gamma)$ which returns the conjunction of the individual formulas in Γ. The function $f(\Sigma, \phi) = \{\sigma | \sigma \in \Sigma \textit{ and } \sigma \textit{ or } \neg\sigma \textit{ appears in } \phi\}$ returns the set of Boolean variables from Σ that appear in ϕ in either a positive or negative form. For example, if $\phi = idle \wedge validID$, $f(\Sigma_{\text{msgs}}, \phi) = \{idle\}$ and $f(\Sigma_{\text{condition}}, \phi) = \{validID\}$.

We take as input the automaton structure for a chart in the form of a symbolic automata structure, A, with an empty final state set. Intuitively, to capture the bad behaviors of a chart, we transform the basic automaton structure to the negative automaton that is capable of detecting safety and liveness errors by yielding accepting cycles in the verification. We do so by adding accept states to the automaton and adding/updating all transitions.

Fig. 3 shows an intuitive description of the outgoing transitions of a state in the transformed automaton. The sets ψ_c, ψ_m and ψ_i (initialized by the algorithm in Fig. 4) are sets of condition, message, and invariant letters used in the outgoing transitions of a given state. There are three

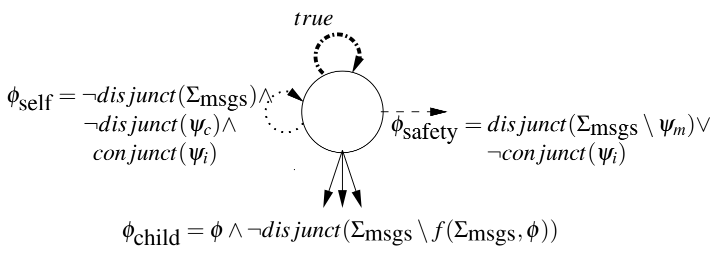

$true$
$\phi_{\text{self}} = \neg disjunct(\Sigma_{\text{msgs}}) \wedge \neg disjunct(\psi_c) \wedge conjunct(\psi_i)$
$\phi_{\text{safety}} = disjunct(\Sigma_{\text{msgs}} \setminus \psi_m) \vee \neg conjunct(\psi_i)$
$\phi_{\text{child}} = \phi \wedge \neg disjunct(\Sigma_{\text{msgs}} \setminus f(\Sigma_{\text{msgs}}, \phi))$

Figure 3: A generic state in the transformed automaton with complete annotations for all types of outgoing transitions 1. ϕ_{self}: self-loop for non-progress, 2. ϕ_{safety}: transition to state q_{safety} for detecting safety errors, and 3. ϕ_{child} transitions to the successor states.

types of transitions that are introduced/updated for each state in the automaton. The ϕ_{safety} transition (dashed edge) leads to the safety state and is responsible for detecting any safety errors. The self-loop (dotted edge), ϕ_{self}, enables the automaton to remain in the current state until an event or condition progresses the automaton to a successor state. The ϕ_{child} transitions (solid edges) lead to the successor states. The dash-dot edge is only added to the first state of the automaton to enable verification of multiple chart instances in a reactive system.

States are marked as accept states in the automaton based on two criteria. First, the safety state is marked as an accept state for detecting safety violations such as duplicate message instances and out of order messages. Second, any state that is not a legal exit point of the chart is marked as an accept state. We now discuss in detail the creation of the transitions and the marking of accept states.

Fig. 4 shows the algorithm for transforming the input automaton. We only present an overview of the algorithm in this version of the paper and refer the reader to the long version for more details. The algorithm has a general Depth First Search (DFS) structure with line 4 enumerating the successors and line 11 making a recursive call for each successor. The algorithm is always invoked for the one initial state of the input automaton to be transformed. Lines 1 - 2 remove any transitions to the exit state q_{exit}. In the automaton shown in Fig. 2(a), the transition from state q_2 to the exit state q_{exit} is removed. Lines 5 - 7 of the algorithm build the sets of variables that are used for messages, invariants, and conditions in the transitions from the current state to the successor states.

Lines 8 - 10 update the transitions to the successor states by first removing the transition and adding a new transition with the updated label. The updated child transition ensures that only the enabled messages, invariants and conditions at a given state can enforce progress in the automaton. For example, the algorithm transforms the transition from state q_1 to state q_2 in Fig. 2(a) from $\phi = jobID$ to $\phi_{\text{child}} = jobID \wedge \neg idle \wedge \neg getData \wedge \neg data \wedge \neg result! \wedge \neg result?$.

Lines 12 - 15 update the self-loop for the current state to ensure that the automaton remains in the current state if no relevant messages are observed. For example, in state q_1 of Fig. 2(b), the self-loop $\neg idle \wedge \neg jobID \wedge \neg getData \wedge \neg data \wedge \neg result? \wedge \neg result!$ is enabled if no message is observed. As mentioned earlier, the first state of the automaton has a self-loop with the $true$ label

```
Algorithm: $TRANSFORM(q)$
1:  for $\forall\delta : (q,\phi,q_{\text{exit}}) \in \Delta$ do
2:      $\Delta \leftarrow \Delta \setminus \{\delta\}$
3:  $\psi_m \leftarrow \emptyset,\ \psi_i \leftarrow \emptyset,\ \psi_c \leftarrow \emptyset$
4:  for $\forall\phi,q' : (q,\phi,q') \in \Delta$ do
5:      $\psi_m \leftarrow \psi_m \cup f(\Sigma_{\text{msgs}},\phi)$
6:      $\psi_i \leftarrow \psi_i \cup f(\Sigma_{\text{invariant}},\phi)$
7:      $\psi_c \leftarrow \psi_c \cup f(\Sigma_{\text{conditions}},\phi)$
8:      $\Delta \leftarrow \Delta \setminus \{(q,\phi,q')\}$
9:      $\phi_{\text{child}} \leftarrow \phi \wedge \neg disjunct(\Sigma_{\text{msgs}} \setminus f(\Sigma_{\text{msgs}},\phi))$
10:     $\Delta \leftarrow \Delta \cup \{(q,\phi_{\text{child}},q')\}$
11:     $TRANSFORM(q')$
12: for $\phi : (q,\phi,q) \in \Delta$ do
13:     $\Delta \leftarrow \Delta \setminus \{(q,\phi,q)\}$
14: $\phi_{\text{self}} \leftarrow \neg disjunct(\Sigma_{\text{msgs}}) \wedge \neg disjunct(\psi_c) \wedge conjunct(\psi_i)$
15: $\Delta \leftarrow \Delta \cup \{(q,\phi_{\text{self}},q)\}$
16: if $\exists\phi,q' : (q,\phi,q') \in \Delta$ and $f(\Sigma_{\text{main}},\phi) \neq \emptyset$ then
17:     $\phi_{\text{safety}} \leftarrow disjunct(\Sigma_{\text{msgs}} \setminus \psi_m) \vee \neg conjunct(\psi_i)$
18:     $\Delta \leftarrow \Delta \cup \{(q,\phi_{\text{safety}},q_{\text{safety}})\}$
19:     if $(q,\phi,q') \in \Delta_{\text{hot}}$ then
20:         $F \leftarrow F \cup q$
21: return($A$)
```

Figure 4: Algorithm for building a negated automaton from an input LSC automaton.

and the final state of the automaton has no self-loops. These special cases are not shown in the transformation algorithm in Fig. 4.

If the current state q contains a main chart transition (labels of transitions to successor states are members of the main chart alphabet Σ_{main}), then lines 16 - 18 of the algorithm add a safety transition to the safety state q_{safety}. The safety transition enables the automaton to detect message order violations or duplicate messages. For the automaton shown in Fig. 2(a), state q_1 contains a single transition for the *jobID* message. Since *jobID* is a member of the main chart alphabet ($jobID \in \Sigma_{main}$) a safety transition needs to be added. The safety transition for state q_1, $idle \vee getData \vee data \vee result? \vee result!$, detects the presence of any message except the one allowed message *jobID*. Because states with no main chart transitions can not violate the chart, no safety transitions are added to them.

Lines 19 - 20 of the algorithm label the current state as an accept state if it belongs to the main chart and contains a hot outgoing transition. The check for main chart transitions is performed on line 16. To check for hot outgoing transitions, each outgoing transition is checked for membership in the Δ_{hot} set (line 19). If all outgoing transitions from a state are cold, the state is not marked as an accept state. In our example, for state q_2, the only outgoing transition corresponds to a cold condition and is not part of the Δ_{hot} set; thus, state q_2 is not marked as an accept state. On the other hand state q_1 is marked as an accept state because it has one successor transition that corresponds to the hot message *jobID*.

We now state the theoretical results of the presented transformation. We first show that for any main chart state in the automaton at least one transition is enabled for any arbitrary input (i.e. the transition relation for main chart states is total). Having enabled transitions guarantees that the automaton does not ignore any inputs which could cause violations or progress in the chart. To conserve space, all proofs have been omitted from this version of the paper but are available in the long version of the paper.

Lemma 1 *For all states containing outgoing main chart transitions, the transition relation is total. Formally, given a state q with a main chart transition* $\left(\bigvee_{\forall \phi_i, q_i : (q, \phi_i, q_i) \in \Delta} \phi_i\right) = true$.

Lemma 1 is only applicable to states containing main chart transitions. Regarding states that do not contain main chart transitions, the safety transition ϕ_{safety} is not added, resulting in an incomplete transition relation. Since these states are responsible for detecting the completion of the pre-chart and not for detecting violations or errors, the incompleteness of the transition relation does not affect the correctness of observing the pre-chart. Our next result states that for all states except the first state of the automaton, the transition relation is deterministic. The transformed automaton is non-deterministic only in the first state (self-loop annotated with $true$) to accommodate for the global verification of every possible instance of the chart in the system. Non-deterministic automata as used in [KTWW06] result in error traces that have to be validated using full LTL verification, which has been shown to be impractical for LSCs [KM07]. Using deterministic automata guarantees that any reported errors are in fact valid errors in the system.

Lemma 2 *For states q in the transformed automaton (except the initial state), the transition relation is deterministic. Formally,* $\forall q \in Q, \forall \phi_i, \phi_j : (q, \phi_i, q_i) \in \Delta \wedge (q, \phi_j, q_j) \in \Delta, (\phi_i \wedge \phi_j) = false$.

The above result guarantees that for any given input to the transformed automaton (except the first and last state) exactly one transition is ever enabled. We now state our primary result for the transformed automaton. Intuitively, we show by application of Lemma 1 and Lemma 2 that the transformed automaton accepts only those words that are not accepted by the LSC and is capable of detecting all behaviors in a system that violate the LSC. We assume that the automaton created detects all pre-chart instances correctly.

Theorem 1 *The automaton, A, generated by the transformation algorithm in Fig. 4 for a given LSC, $SPEC$, defined over an alphabet $\Sigma_{SPEC} \subseteq \Sigma$, reads exactly the complement of the language of the $SPEC$. Formally,* $\forall \theta = \theta_0 \theta_1 \theta_2 \ldots$

$$[\theta \in L(SPEC) \implies \theta \notin L(A)] \wedge [\theta \notin L(SPEC) \implies \theta \in L(A)].$$

where $L(A)$ and $L(SPEC)$ are the languages of the transformed automaton and the $SPEC$.

4.1 Verification Approach

For explicit state model checking, verification of a system against the specification is performed in the usual manner. The composition of the system and transformed LSC automata is computed on-the-fly and checked for accepting cycles using the Double Depth First Search (DDFS) algorithm. If the DDFS algorithm does not discover any accepting cycles, the system implements the safety and liveness behaviors as described in the chart. For symbolic model checking, we first label accept states as fair states in the composition of the system and transformed LSC automata. This automaton is then verified against the ACTL property $\mathbf{EG}(true)$, which searches for fair Strongly Connected Components (SCCs) reachable from the initial state. Any reported SCCs are violations of the specification.

5 Analysis

The verification approach presented in [KTWW06] utilizes at least two and in the worst case three algorithms to completely verify a system against an LSC. If reachability analysis followed by ACTL verification fails to produce a significant result (proof of correctness or a violation) the system is verified against an LTL formula generated from the LSC specification [TW06]. Compared to the verification approach of [KTWW06], the new verification approach presented in this paper only performs one verification run of comparable complexity as the reachability analysis and ACTL verification in the approach of [KTWW06]. In the average case the total verification cost is reduced by a factor of two and in the best case (worst case in old approach) by a factor of three or more.

One side effect of using the negative automaton is the inability to verify multiple instances of a chart with cold construct violations. For example, if in our example system the *Node* receives *jobID* but is unable to validate *jobID*, the cold condition *validID* is never observed and the chart automaton will remain in state q_2. This is not an error since state q_2 is a non-accepting state waiting to observe the cold condition *validID*. If *Node* restarts the job acquisition by sending the *idle* message to the *Scheduler*, the safety transition from state q_2 to q_{safety} is taken. Consequently, a false error will be reported (duplicate message). Generally speaking, if in one instance of the chart a cold construct is never observed, no future instances of the chart can be observed in a given trace. This drawback can be limiting for highly reactive and iterative systems with multiple instances in a single trace. A solution is being investigated as future work.

6 Results

We briefly discuss our experiments and results in this section. For a detailed presentation we refer the reader to the long version of the paper. We create models with multiple communicating processes and test them against highly concurrent worst case specifications as described in [KTWW06]. All specifications are named A$c \times m$ where c and m are the number of co-regions and messages in each co-region respectively.

We first test the scalability of our approach in the symbolic model checking domain and compare it to the results presented in [KTWW06]. Table 1 shows a subset of the results for verifying the *abp* model using the NuSMV model checker. In general, our verification approach performs twice as fast as the approach presented in [KTWW06] and we scale to specification sizes that were unobtainable using the verification approach in [KTWW06]. We also test the scalability of our approach in explicit state model checking using the SPIN model checker. Table 2 shows a subset of the results for verifying the *plain* and *soko* models. Our approach performs better and scales to larger specifications when compared to the approach of [KM07].

Table 1: Results for the traditional and improved verification approaches using NuSMV.

Specification	Traditional Verification						Improved Verification	
	Reachability		ACTL		Total			
	States	Time	States	Time	States	Time	States	Time
A3x5	1.01616e+06	34	1.47142e+07	35	15730360	69	1.41696e+07	34
A3x6	1.01616e+06	237	1.01616e+06	239	2032320	477	471552	251
A3x7	879408	1568	879408	1562	1758816	3130	521504	1550

Table 2: Results for the improved verification approach using SPIN.

Specification	Model	Without Errors			With Errors		
		States	Memory	Time	States	Memory	Time
A7x6	soko	97500	17.216	125	89323	16.397	125
	plain	406	7.385	123	406	7.385	124
A8x6	soko	97500	18.491	214	89323	17.672	210
	plain	406	8.661	216	406	8.661	215
A9x6	soko	97500	20.104	325	89323	19.285	344
	plain	406	10.274	335	406	10.274	334

7 Conclusions and Future Work

The presented LSC to automaton transformation algorithm allows us to verify a system against an LSC using only language containment with readily available tools. Compared to past approaches, this approach only requires one verification run of comparable complexity as opposed to three verification runs for any arbitrary LSC. Further, we prove that the generated automaton can detect all safety and liveness violations in a system and empirically show the effectiveness of the approach. For future work we are investigating the use of LSCs for automated environment generation to test individual interfaces in a system. We are also investigating the possibility of extending the transformation algorithm to constructs such as overlapping chart instances, Kleene star and multiple instance detection with the presence of cold constructs (as discussed earlier).

Bibliography

[AY99] R. Alur, M. Yannakakis. Model Checking of Message Sequence Charts. In *CONCUR99: Proc. of the 10th Int. Con. on Concurrency Theory*. Pp. 114–129. Springer-Verlag, London, UK, 1999.

[BDK+04] M. Brill, W. Damm, J. Klose, B. Westphal, H. Wittke. Live sequence charts: An introduction to lines, arrows, and strange boxes in the context of formal verification. In *SoftSpez Final Report*. Lecture Notes in Computer Science 3147, pp. 374–399. Springer, 2004.

[BH02] Y. Bontemps, P. Heymans. Turning high-level live sequence charts into automata. In *Proc. of "Scenarios and State-Machines: Models, Algorithms, and Tools" (SCESM02) Workshop of the 24th Int. Conf. on Software Engineering (ICSE 2002)*. Orlando, FL, May 2002.

[BHK03] Y. Bontemps, P. Heymans, H. Kugler. Applying LSCs to the specification of an air traffic control system. *Proc. of the 2nd Int. Workshop on Scenarios and State Machines: Models, Algorithms and Tools (SCESM03), at the 25th Int. Conf. on Software Engineering (ICSE03), Portland, OR, USA*, 2003.

[DH99] W. Damm, D. Harel. LSCs: Breathing life into message sequence charts. In *Proc. of the 3rd Int. Conf. on Formal Methods for Open Object-Based Distributed Systems (FMOODS99)*. P. 451. Kluwer, B.V., Deventer, The Netherlands, 1999.

[DK01] W. Damm, J. Klose. Verification of a radio-based signaling system using the STATEMATE verification environment. *Formal Methods in System Design* 19(2):121–141, 2001.

[HK01] D. Harel, H. Kugler. Synthesizing state-sased object systems from LSC specifications. In *CIAA00: Revised Papers from the 5th Int. Conf. on Implementation and Application of Automata*. Pp. 1–33. Springer-Verlag, London, UK, 2001.

[IT93] R. ITU-T. 120: Message sequence chart (MSC). *Telecommunication Standardization Sector of ITU, Geneva*, 1993.

[KHG05] C. Knieke, M. Huhn, U. Goltz. Modeling and simulation of an automotive system using LSCs. *Proc. of the 4th Int. Workshop on Critical Systems Development Using Modelling Languages (CSDUML05)*, 2005.

[Klo03] J. Klose. *Live sequence charts: A graphical formalism for the specification of communication behavior*. PhD thesis, Fachbereich Informatik, Carl Von Ossietzky University, 2003.

[KM07] R. Kumar, E. Mercer. Improving translation of live sequence charts to temporal logic. In *Proc. of the 7th Int. Conf. on Automated Verification of Critical Systems (AVoCS07)*. Pp. 183 – 197. 2007.

[KTWW06] J. Klose, T. Toben, B. Westphal, H. Wittke. Check it out: On the efficient formal verification of live sequence charts. *Proc. of the 18th Int. Conf. on Computer Aided Verification (CAV06), LNCS* 4144:219–233, 2006.

[KW01] J. Klose, H. Wittke. An automata based interpretation of live sequence charts. In *TACAS 2001: Proc. of the 7th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*. Pp. 512–527. Springer-Verlag, London, UK, 2001.

[SD05] J. Sun, J. S. Dong. Model checking live sequence charts. In *ICECCS05: Proc. of the 10th IEEE Int. Conf. on Engineering of Complex Computer Systems (ICECCS05)*. Pp. 529–538. IEEE Computer Society, Washington, DC, USA, 2005.

[TW06] T. Toben, B. Westphal. On the expressive power of LSCs. In *SOFSEM 2006: 32nd Conf. on Current Trends in Theory and Practice of Computer Science, Měřín, Czech Republic, January 2006*. Volume 2, pp. 33–43. Institute of Computer Science AS CR, Prague, 2006.