Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 250 (2009) 33–48
www.elsevier.com/locate/entcs

Verifying Communication Protocols Using Live Sequence Chart Specifications

Rahul Kumar[1] Eric G Mercer[2]

Computer Science Department, Brigham Young University,Provo, Utah 84606, USA

Abstract

The need for a formal verification process in System on Chip (SoC) design and Intellectual Property (IP) integration has been recognized and investigated significantly in the past. A major drawback is the lack of a suitable specification language against which definitive and efficient verification of inter-core communication can be performed to prove compliance of an IP block against the protocol specification. Previous research has yielded positive results of verifying systems against the graphical language of Live Sequence Charts (LSCs) but has identified key limitations of the process that arise from the lack of support for important constructs of LSCs such as Kleene stars, subcharts, and hierarchical charts. In this paper we further investigate the use of LSCs as a specification language and show how it can be formally translated to automata suitable for input to a model checker for automatic verification of the system under test. We present the translation for subcharts, Kleene stars, and hierarchical charts that are essential for protocol specification and have not been translated to automata before. Further, we successfully translate the BVCI protocol (point to point communication protocol) specification from LSC to an automaton and present a case study of verifying models using the resulting automaton.

1 Introduction

System on Chip (SoC) designs are fast moving towards a development environment that incorporates third party Intellectual Property (IP) cores and blocks. Due to the use of such heterogeneous IP cores, multiple communication protocols are required to achieve the desired interactions, behavior, and functionality. With this diverse development environment comes not only the burden of verifying the system under development, but also the third party modules and communication protocols, to ensure the correctness and compliance of the complete system with respect to the system specification. This need is especially important for vendors looking to market and promote their products in new markets and development environments. To reduce the verification costs and redundancy of verification (verified twice: once

[1] Email:rahul@cs.byu.edu
[2] Email:egm@cs.byu.edu

doi:10.1016/j.entcs.2009.08.016

by the IP core developer and once again by the integrator), IP cores are often verified against commonly accepted standards and specifications to provide compliance results that can be easily utilized in an integration environment. A significant issue that hinders the process is the lack of an accepted specification language that can be formally integrated into the verification environment.

Traditionally, English has been used as the specification language for describing communication protocols. Due to the ambiguous and informal nature of English, in our experience, it has proven to be an inefficient specification language for use in a formal verification environment. Other specification languages based on temporal logic have also been used to specify correctness requirements of systems. Due to the complex nature of the temporal logics and the lack of support in all verification tools, these specifications tend to be limited in their use and applicability. Although specification patterns developed in the past do help, some aspects of creating a complete specification for an arbitrary verification are always unique and have to be created ground up; thus, re-enforcing the difficulty of using temporal logics as a specification language.

Other research has also investigated the use of graphical languages such as Live Sequence Charts (LSCs) for specifying communication protocols, and have reported positive and encouraging results [5]. We choose LSCs as our specification language because of their direct applicability to specifying communication protocols that primarily describe inter-process communication. Additionally, their graphical and intuitive nature makes them extremely usable for everyone involved in the development process and not only experts of formal verification.

In the past, LSCs have been used both as a specification and a modeling language. Because of their inherent ability to specify communication patterns without data information, we choose to use LSCs as a specification language describing the correctness requirements of a system. Previous work in the area of using LSCs as a specification language in a formal verification environment has been effective in exploring a verification approach but has failed to provide a comprehensive solution that supports the entire LSC grammar, which includes constructs such as Kleene stars, subcharts and hierarchical charts. We show how a communication protocol implementation can be formally verified against an LSC specification by providing translations of the entire LSC grammar to an automaton that is similar in nature to a *never claim* generated by SPIN [8]. This automaton can then be used directly as input to a model checker for verification of the system under test. Further, we provide a case analysis where the entire Basic Virtual Component Interface (BVCI) protocol is translated to an automaton and Promela models are verified against the translated automaton [5].

Using a graphical specification language targeted towards communication protocols provides the inherent advantage of rapid development of specifications that are intuitive and useful throughout the development cycle of the product. Since LSCs can be used both as a modeling and a specification language, they provide a common medium for verifying requirements as well as systems. Using the translation to automaton as presented in this paper, the specification can now be applied

more directly in a formal verification approach. Additionally, this approach does not require specialized tools and algorithms to be applied for formal verification of a system. It relies only on the synchronous composition of the model with the specification for detection of accepting cycles using the Double Depth First Search (DDFS) algorithm or a simple ACTL formula (for labeling algorithms) [12]. Doing so, allows the technique to retain the advantage of any custom model abstractions or state space reduction techniques supplied by the model checker.

The paper is organized as follows. Section 2 presents an overview of related work in the field of LSCs and verification using LSCs. Section 3 gives an overview of the LSC constructs and provides an example of an LSC that is explained in detail. Section 4 presents an overview and examples of the LSC to automaton translation method. Section 5 discusses the case study and presents results of verifying Promela models against the BVCI LSC specification followed by conclusions in Section 6.

2 Related Work

LSCs are an extension to Message Sequence Charts (MSCs) [9]. The most significant addition to the MSC language is the introduction of *liveness* or *provisional* behavior that distinguishes between mandatory and optional behavior [4]. Additionally, the LSC language also provides constructs such as temperatures, subcharts, and precharts that enable the user to describe behaviors that could not have been described in MSCs. Protocol Live Sequence Charts (PLSCs) are an extension to LSCs that are targeted to describing protocols [5].

LSCs have been used to model and specify a variety of systems such as air traffic control systems [3], radio based communication systems [6], and train systems [2]. Their use in these case studies has shown their effectiveness in specifying and verifying complex behaviors of a system. LSCs and PLSCs have also been used in the past to specify SoC communication protocols and formally verify aspects of the protocol on the system [5]. Additionally, they have also been used for automatic synthesis of systems as well [7].

Recently, LSC based verification techniques have been gaining significant attention. One aspect of LSC related verification deals with verifying properties on the LSC specification itself [1,16]. In this case, the LSC is used as the model.

Another aspect of LSC based verification deals with the verification of systems against the LSC specifications. Two primary methods have been proposed to perform verification of systems against LSCs. The first deals with temporal logic. One approach converts the LSC specification to multiple small temporal logic properties that are verified on the system [5]. These individual properties are easily verified on a system but are insufficient to establish a formal relationship between the specification and implementation itself. Other approaches translate the complete chart to temporal logic, which is then used as the specification input to a model checker such as SPIN or NuSMV [13,11]. The primary limitation of these approaches is the exponential explosion encountered in the generated temporal logic formula (number of nested temporal operators), which severely reduces the scalability of the approach.

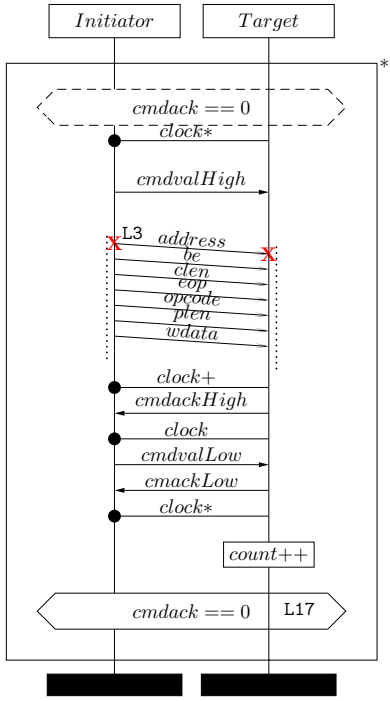


Fig. 1. Example LSC showing the *Normal Request* scenario.

Additionally, the lack of support for translating the complete grammar of LSCs to temporal logic is a great limiting factor in the applicability of the approaches.

The second method for verifying systems against LSCs does so by converting the LSC to an automaton and using the automaton in language containment based verification techniques [12]. This method supports a greater subset of the LSC grammar and scales to much larger specification sizes. Although the verification results and performance using the automaton approach for verifying systems are very promising, the research does not deal with constructs such as subcharts, hierarchical charts, and Kleene stars, which are essential to the specification and verification of SoC interface and communication protocols. The work presented in this paper innovates upon previous work by extending the translation of LSCs to the complete grammar of LSCs.

3 Live Sequence Charts

The LSC language provides constructs to express behavior of systems and individual processes with relative ease and intuitiveness. The primary advantage lies in its graphical yet formal nature. Fig. 1 shows an example LSC for the *Normal Request* subchart in the BVCI protocol [5]. We use this example as our basis for introducing the constructs and semantics of the LSC language.

Processes or Instances: Processes are drawn with rectangular *instance heads* that denote the start of the processes. A vertical line originating from the instance head signifies the *life-line* of the process and ends in a filled rectangle, which termi-

nates the respective process. The example LSC describes the interaction between the *Initiator* and *Target* processes.

Locations: The life-line of each process is marked with locations that are points where events and other constructs may be described. Locations are unique to each process and start at location `L1`. For each new event or construct placed on the process life-line, the location number is incremented for the respective process. For example, the *address* message is sent from the *Initiator* process at `L3` and *Target* evaluates the $cmdack == 0$ post condition at `L17`.

Messages: Messages are a form of communication between processes in the LSC. Each message has a sender and receiver process attached to it. Messages are annotated with a message label that identifies the message. Messages can be *simultaneous* or *asynchronous*. Simultaneous messages are drawn with a solid arrow head and occur instantaneously when both the sender and receiver are ready for the communication. Asynchronous messages are drawn with an open arrow head and can be received any time after sending (we force the send event to occur before the receive event). In the example LSC, the *address* message is an asynchronous message and the *cmdackHigh* message is a synchronous message.

Conditions: Conditions are placed in the chart by drawing hexagons around the life-lines of processes evaluating the condition. The condition label describes a predicate that must be satisfied at the current location(s) of the process(es). Conditions spanning multiple process life-lines act as synchronizing points for the involved processes and the condition is not evaluated unless all the processes are at the respective condition locations. Conditions attached to a message are called *bonded conditions*. Conditions placed on their own location and not attached to a message in the chart are called *non-bonded* conditions [10]. Non-bonded conditions are evaluated continuously until they are satisfied. In our example LSC, all conditions (the $cmdack == 0$ precondition and the $cmdack == 0$ postcondition) are non-bonded. *Invariants* are conditions spanning over multiple locations in the chart.

Coregions: Coregions are drawn with a dashed vertical line parallel to the life-line of a process and are used to describe behavior that can occur in any order. All messages in the dashed vertical line (*address*, *be*, *clen*, etc.) next to the *Initiator* and *Target* processes are in a coregion.

Simultaneous regions: Simultaneous regions describe events that occur at the exact same time. Dots are drawn on locations to indicate simultaneity of events.

Actions: The LSC language also provides the action construct that allows a process to perform an action on its local or global variables. For example, variables may be incremented, decremented or assigned a value at certain points on the life-line of a process. In the example LSC of Fig. 1, the $count++$ action is performed by the *Target* process before the postcondition is evaluated. Currently, actions do not translate to a state in the automaton generated from the LSC specification. Although, it is possible to check the effect of an action by automatically generating a condition in the LSC to ensure that the action has been performed successfully.

Prechart: The prechart is drawn with a dashed hexagon encompassing the instance heads and connects to the *main* body of the chart that is described in the

solid rectangle following the prechart. The prechart describes the behavior of the system under which the main body of the chart is to be observed. The prechart can also be substituted with a single activation condition.

Main chart: The main chart of the LSC specifies the behaviors described in the rectangle following the prechart. The main chart can be either *existential* or *universal*. Universal charts, drawn with a solid rectangle, specify behavior that must be satisfied by the system every time the prechart is satisfied. Existential charts are drawn with a dashed rectangle and specify behavior that the system must exhibit at least once when the prechart is satisfied. In the example LSC of Fig. 1, the main chart is a universal chart.

Subcharts: Subcharts are LSC charts that can be included within the body of a larger main chart. They are usually not preceded by a prechart. When a subchart B is included within the main chart of A, chart A is at a higher scope than subchart B. Subcharts in conjunction with conditions and Kleene stars can be used to create control and looping structures such as *if-then* and *while* blocks. The example chart shown in Fig. 6(a) shows one main chart that contains a subchart as well. The semantics of subcharts are discussed in greater detail in Section 4.2.

Temperatures: Temperatures in LSCs can be assigned to messages, conditions, and locations. A *hot* temperature is depicted by using a solid line to draw the construct and specifies behavior that must be satisfied by the system. A *cold* temperature is drawn using a dashed line for the construct and specifies behavior that may be satisfied. If a cold message is never observed, the LSC waits at the current location for the message. If on the other hand, the construct after the cold message in the chart is observed, the LSC progresses to the location after the cold message. Bonded cold conditions do not affect the LSC execution. If the condition is not satisfied, an error is *not* reported and the LSC exits the current scope to a higher scope. If no higher scope exists, the LSC exits completely. In the case of a non-bonded cold condition, the LSC waits indefinitely at the current location for the condition to be satisfied and can only exit the current scope if a construct at a higher scope is observed. It is not possible for the LSC to move to a location after the non-bonded cold condition within the same chart until the non-bonded cold condition is satisfied. If no higher scope exists, the LSC waits indefinitely for the non-bonded cold condition to be satisfied. For the example chart shown in Fig. 6(a), if the non-bonded cold condition p is not satisfied, then the LSC waits at the current location until either p is satisfied or a b is observed. All constructs in the example LSC in Fig. 1 are hot except for the activation condition $cmdack == 0$.

Kleene star: The Kleene star construct, '$*$', is used to represent multiplicity where the associated chart/message can occur zero or more times (finite). In our example LSC, the clock signal following the $cmdack == 0$ condition can occur as many times as required before the coregion messages are observed. A variation of the Kleene star is the '+' symbol that forces at least one (and allows more than one) occurrence of the associated construct.

Hierarchical charts: Hierarchical charts are constructed using individual LSCs and are useful for creating specifications that require control flow. Hierarchical LSCs

are similar to LSC subcharts and high level MSCs as described in [14]. Fig. 7(a) shows an example of hierarchical LSCs where A, B, and C are individual LSCs joined together to form a hierarchical LSC.

We have presented the entire set of LSC constructs that are currently supported by our LSC to automaton translation. Apart from the listed constructs, the chart also induces a natural partial order for all constructs along each instance line. Intuitively, instances evolve in the downward direction and are blocked until an event on their life-line occurs. For the example chart shown in Fig. 1, the chart is entered when the $cmdack == 0$ condition is satisfied. After the precondition is satisfied, multiple clock signals may occur before the $cmdvalHigh$ message should be observed followed by the coregion. After all constructs as described in the LSC are observed, the $Target$ process increments the *count* variable and waits for the $cmdack == 0$ condition to be satisfied.

Additionally, we incorporate the *delayed choice* semantics when dealing with subcharts, hierarchical charts, and cold constructs. The delayed choice semantics allow the chart to resolve a choice by waiting for relevant input before committing to a certain path in the LSC. Since we are using the chart as a specification language rather than a modeling language, delayed choice semantics help avoid non-determinism (reduce false positives). If the LSC were to be used as a model rather than a specification, delayed choice semantics would be removed to allow non-determinism in the model.

In our research, we deal with all the described constructs of LSCs with the following restrictions: (a) overlapping instances of charts are not permitted, (b) the prechart and the main chart should have a disjoint set of messages and conditions (to avoid overlapping instances of charts), and (c) only one subchart can be enabled at a given time. These limitations have been introduced to simplify the LSC to automaton translation process and remove any non-determinism. Since most specifications in general do not require overlapping charts and instances, the limitations do not affect the applicability of the results.

4 LSC to Automaton Translation

Our verification approach uses LSCs as the specification and verifies the system by detecting accepting cycles on the synchronous composition of the system automaton and the *negative automaton* of the LSC. The negative automaton of the LSC is the automaton that enables detection of unwanted behaviors in the system (using accept cycles recognized by the LSC automaton). The automaton is similar in nature to the never claim used in SPIN and has been shown to be an effective method of using LSCs for verification [12]. We first present an overview of the LSC to automaton translation for basic constructs as discussed in [12] and then present the translation for extended constructs that have not been explored in previous work: the Kleene star operator, subcharts, and hierarchical charts. To conserve space, we restrict our discussion to universal main charts only.

Before we discuss the LSC to automaton translation, we introduce some neces-

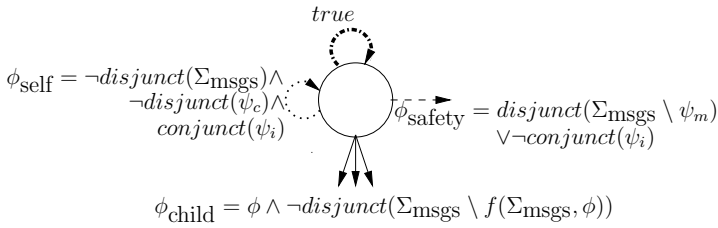


Fig. 2. Generic state of the LSC to automaton translation.

sary formalism. We use *Symbolic automata*, an extension of Büchi automata, that allow 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 $\Delta \subseteq Q \times \rho \times Q$ is the transition relation. A transition $(q, \rho, q') \in \Delta$ represents the change from state q to state q' when the formula ρ 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, $\Sigma_{\text{msgs}} = \{address, opcode, clen, \ldots\}$ and $\Sigma_{\text{conditions}} = \{cmdack == 0\}$. The set $\Sigma_{\text{main}} = \{address, opcode, clen, \ldots\}$ 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 formulas $\Gamma = \{\rho_0, \rho_1, ..., \rho_n\}$ we define the function $disjunct(\Gamma)$ to return the disjunction of the individual formulas in Γ and the function $conjunct(\Gamma)$ to return the conjunction of the individual formulas in Γ. The function $f(\Sigma, \rho) = \{\sigma | \sigma \in \Sigma \; and \; \sigma \; or \; \neg\sigma \; appears \; in \; \rho\}$ returns the set of Boolean variables from Σ that appear in ϕ in either a positive or negative form. For example, if $\rho = a \wedge b$, and b is a condition predicate, $f(\Sigma_{\text{msgs}}, \rho) = \{a\}$ and $f(\Sigma_{\text{condition}}, \rho) = \{b\}$. Additionally, the ψ_m, ψ_c and ψ_i sets contain the message, condition and invariant predicates appearing in the current state of the automaton (predicates in the current cut).

The automaton of the chart is obtained by exploring every possible *cut* through the chart. A cut through a chart represents the current state of the chart as specified by the location of each process in the chart and the state of the variables of the chart. For the example LSC shown in Fig. 1, the **X** marks on each instance line represent a cut through the chart. At this cut the *Initiator* and *Target* processes are at the beginning of their coregion ready to send/receive any of the messages in the coregion.

From a given cut, enabled transitions lead to successor cuts. The enabled transitions correspond to the set of events that can occur from a given cut without violating the partial order induced on the events by the instances in the chart. Each unique cut of the LSC corresponds to a unique state in the automaton. The unwinding algorithm as presented in [10] provides a method to unroll the LSC and

all possible cuts of the LSC; thus, it gives the basic structure of the LSC automaton. This basic structure of the automaton is then transformed to a negative automaton using the transformation algorithm presented in [12]. It should be noted that the unwinding algorithm presented in [10] does not support Kleene stars, subcharts, and hierarchical charts. Additionally, the basic structure generated from the LSC is not as efficient as the transformed automaton presented in [12].

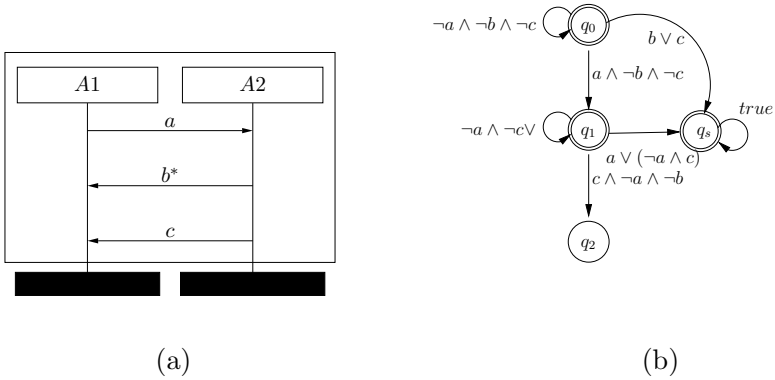
The general structure of the LSC automaton can be split into two parts: the prechart automaton and the main chart automaton. Additionally, a special state in the automaton is the safety state, q_s: an accepting state that contains only one outgoing transition to itself labeled with *true*. The prechart automaton contains only non-accepting states since the prechart is responsible for the detection of the activation condition of the main chart. Additionally, the prechart states do not contain transitions to the safety state since the prechart does not detect errors or incorrect behavior. The first state of the prechart contains a special outgoing transition to itself that is labeled *true* to ensure that all possible instances of the charts in the system are checked for errors (corresponds to globally).

Fig. 2 shows a generic state of the automaton with all possible outgoing transitions. The dotted self-loop in each state is to detect non-progress when no relevant letters are observed (liveness errors). The solid transitions to child states detect the progress through the LSC when relevant letters are observed. Multiple child states occur when concurrency is present in the chart. The dashed transition from main chart states to the safety state allows detection of safety errors (out of order messages, invariants, etc.). Main chart states are marked accept states if at least one progress transition corresponds to a hot construct from the main chart. The final state of the automaton is non-accepting and contains no outgoing transitions. Using the transition labels as shown in the generic state of Fig. 2, the transition relation for each main chart state is proven to be deterministic and total, which is further utilized in the proof of correctness for the translation.

4.1 Kleene Star

Kleene stars can be placed on messages or subcharts to indicate repetition. When a Kleene star is placed on a construct (message or subchart), the construct may be observed in the system zero or more times (finite). We first show how Kleene stars attached to messages are translated to automaton and discuss the translation of subcharts with Kleene stars in Section 4.2.

Fig. 3(a) shows an LSC where message b may occur zero or more times. The corresponding automaton is shown in Fig. 3(b). The first state, q_0, corresponds to the locations in the LSC where message a is yet to occur. The safety transition is enabled if any out of order messages $(b \vee c)$ are observed. The second state q_1 corresponds to the state where the message b can occur repeatedly (finite number of occurrences). To accommodate for this repetition, a new disjunctive clause $b \wedge \neg a \wedge \neg c$ is added to the self-loop (expressions reduce to $\neg a \wedge \neg c$). The modification allows the automaton to remain in state q_1 as long as no relevant messages or the message b is observed repeatedly. The safety transition is also modified to allow



(a) (b)

Fig. 3. Example of translating a message with Kleene star.

multiple occurrences of b. Finally, if c is observed, the automaton moves to state q_2, which is the end of the LSC and the automaton. It should be noted that currently we do not handle the b^*b case.

A variation of the Kleene star construct is the '+' operator that is used to specify that a message should appear at least once. If message b in Fig. 3(a) is changed to have a '+' rather than a '$*$' operator, the corresponding automaton is shown in Fig. 4. The major difference between the Kleene star and '+' operator translation lies in the introduction of an extra state (q_1) to ensure at least one instance of the message b is observed. The state q_1 waits for the first b to be observed and the state q_2 allows an infinite number of b's to be observed. The extra state is introduced by the LSC unwinding algorithm and the transition labels as described earlier are implemented for each state; thus, making the transition relation total and deterministic and allowing us to apply the correctness proof.

We now formalize our translation of Kleene stars to automaton. The transition labels as shown in Fig. 2 are modified to incorporate translation of the Kleene star. We introduce the sets ψ_k and ψ_x corresponding to the messages that are attached with a Kleene star and messages not attached with a Kleene star for a given state. The self-loop is modified to allow the automaton to remain in the current state if no relevant messages are observed, or if the Kleene star messages are observed: $\phi_{\text{self}} = \neg disjunct(\Sigma_{\text{msgs}} \setminus \psi_k) \wedge \neg disjunct(\psi_c) \wedge conjunct(\psi_i)$. Secondly, the safety transition is modified to disable detection of multiple instances of a message and detect all other possible safety errors as follows: $\phi_{\text{safety}} = disjunct(\Sigma_{\text{msgs}} \setminus (\psi_k \cup \psi_x)) \vee conjunct(\{\neg disjunct(\Sigma_{\text{msgs}} \setminus (\psi_k \cup \psi_x)), \psi_x\})$. Using these modified transition labels, we can now prove that the transition relation for each main chart state is deterministic and total.

Lemma 4.1 *For all states containing outgoing main chart transitions, the transition relation is deterministic and total. Formally, given a state q with a main chart transition:* $\left(\bigvee_{\forall \phi_i, q_i : (q, \phi_i, q_i) \in \Delta} \phi_i\right) = true$, *and* $\forall q \in Q, \forall \phi_i, \phi_j : (q, \phi_i, q_i) \in \Delta \wedge (q, \phi_j, q_j) \in \Delta, (\phi_i \wedge \phi_j) = false$.

By proving that the transition relation for each main chart state is total and

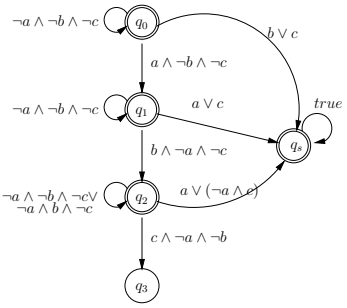


Fig. 4. Translation of the plus operator on a message.

deterministic, we can now show that all safety and liveness errors are detected by the generated automaton. To conserve space, proof details are omitted from this version of the paper.

Theorem 4.2 *The automaton, A, generated 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 automaton and the SPEC.

4.2 Subcharts

We now focus on translating subcharts to automaton. Fig. 5(a) shows an example of an LSC with three subcharts that start with the letters x, y, and z respectively. To conserve space we do not show the entire contents of the subcharts, but focus on the start letter of each subchart. Given the example LSC of Fig. 5(a), the corresponding automaton translation is presented in Fig. 5(b).

To translate a chart that contains subcharts, each subchart is first individually translated to an automaton. After each subchart has been translated to its respective automaton, the automata are combined into one large automaton using the scheme presented in Fig. 5(b). The automaton for the LSC moves from the initial state q_0 to the automaton of chart A when an x is observed (automata combined using standard sequential composition) [15]. After subchart A has been observed successfully, the automaton moves to subchart B and so on.

The dashed transitions in the automaton are introduced to incorporate delayed choice semantics. Such dashed transitions are introduced from every possible legal exit (last state and states corresponding to cold constructs) of a subchart to the entry points of other subcharts or higher scopes. For example, if at a legal exit of subchart A, the letter y is observed, progress is made by exiting chart A and entering chart B. Similarly, from a legal exit of chart A, progress can be made to the beginning of chart C by observing a z. The dash-dot transitions from the legal exits of B to the beginning of B are introduced to incorporate delayed choice for

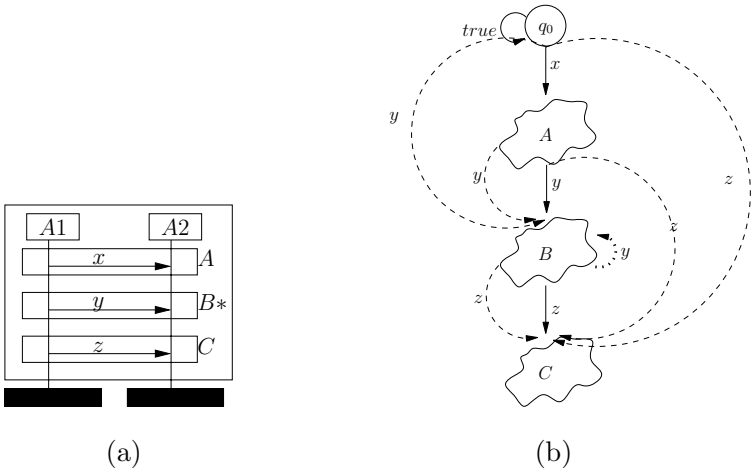


Fig. 5. Example of translating an LSC with multiple subcharts.

the Kleene star attached to subchart B. They ensure that a new instance of B can be observed from an legal exit of B.

Fig. 6(a) shows a subchart with an attached Kleene star. The corresponding automaton (without safety transitions) is shown in Fig. 6(b). At state q_0 the automaton expects to see the message a. Once a has been observed, the automaton moves to state q_1 where it either expects to enter the subchart (waiting for condition p to be satisfied) or to move to the location after the subchart when message b is observed (using the dashed transition from q_1 to q_4). If the subchart is entered, message x is observed in the normal manner. At state q_3, message y is observed and the outgoing transition depends on the Kleene star. As a Kleene star is attached to the subchart, observing message y leads the automaton back to state q_1 using the dotted transition. If the Kleene star did not exist, the automaton would move to state q_4 after observing y. After y has been observed, the automaton waits for b.

To facilitate the translation of subcharts from LSC to automaton, the unwinding algorithm is modified to keep track of the start and end states of each subchart automaton as well as all the legal exits of a subchart. The unwinding algorithm then performs sequential composition of the individual subchart automaton with the main chart using the start and end states of each subchart automaton to create the skeleton automaton for the entire main chart. Once the skeleton automaton for the main chart has been created, for each legal exit of a subchart (recorded by unwinding algorithm), extra child transitions are added to possible future states and the self loop and safety transition labels are modified to ensure that the transition relation remains deterministic and total. Since each transition from a legal exit of a subchart state to a possible future state is considered a child transition, the transition labels shown in Fig. 2 require no change, rather, the creation of the individual sets ψ_k, ψ_m, and ψ_x is modified to include the letters that can lead to higher scope executions. Using the deterministic and total transition relation for each main chart state we can again apply the proof of correctness of Theorem 4.2. To conserve space, the details have been omitted from this version of the paper.

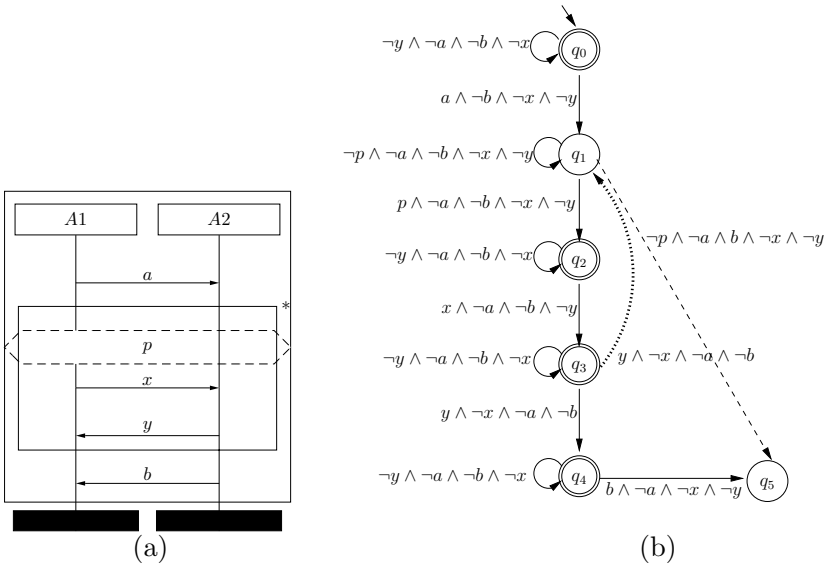


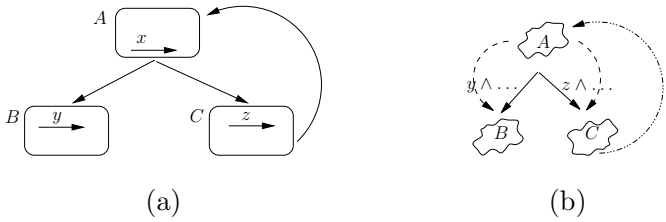
Fig. 6. Example of translating subcharts with attached Kleene star.

Subcharts that occur concurrently with other constructs in the chart are first combined using standard parallel composition of automata (all possible inter-leavings are explicitly expressed) and then combined with the rest of the main chart automaton. For example, if subcharts A and B are concurrent with each other, the automata for the subcharts are composed in parallel and the last state of the automaton is used for performing sequential composition with the automaton of subchart C. Performing the parallel composition is exponential in the worst case.

4.3 Hierarchical Charts

Hierarchical charts allow individual LSCs to be joined together sequentially. The construct is particularly useful in combining large individual LSCs into a visually concise LSC incorporating control flow and choice. Fig. 7(a) shows an example of a hierarchical chart. A, B, and C are individual charts and either chart B or C can be executed after chart A has completed execution. After chart C has completed execution, the hierarchical LSC moves back to chart A. Message x is the final message of chart A and messages y and z are the first messages of charts B and C respectively (activation). The corresponding automaton of the hierarchical chart is shown in Fig. 7(b). To conserve space, we only show the general structure of the automaton and the hierarchical chart.

From the last state of chart A, the automaton has the option of moving to the first state of chart B or the first state of C. If a y is observed, the automaton moves to the automaton of chart B and if a z is observed it moves to the automaton of C. The final state of chart A is not accepting, since no behavior must be observed at this point. The automata are joined together using standard sequential composition. Since chart C always moves back to chart A, a dash-dot transition is introduced from the last state of chart C's automaton to move back to the first state of chart



(a) (b)

Fig. 7. Translating hierarchical charts to automaton.

A. Additionally, the dashed transitions from A to B and A to C are introduced to handle cold constructs. A similar set of transitions is introduced from C to A for cold constructs in C. These transitions can only be introduced to the successor charts of a given chart. For example, such transitions are not introduced from chart B to chart A or C since B has no successors. In the presence of multiple messages at the end of a chart, the translation is always guaranteed to have a final state, which is used as the starting point for joining successor charts.

It should be noted that in accordance with delayed choice semantics the minimal common prefix is chosen to identify the next chart that is to be executed (one message in the example, but could be more than one when complex precharts are specified). In the case of complex precharts (more than one message/condition), each legal exit of a chart leads to a new instance of prechart detection where it is possible to detect progress in the prechart of a successor chart or in the current chart itself (by observing the cold construct). For multiple successor charts, multiple prechart detections are introduced from each legal exit of a chart.

5 Case Study: BVCI Protocol Verification

The Basic Virtual Component Interface (BVCI) protocol is part of the Virtual Component Interface (VCI) standards family that was developed to specify point to point communication protocols. We use the BVCI protocol as our specification for case analysis because of the complex nature of the specification as well as the past research that has been performed on verifying systems against the BVCI protocol [5]. We now describe our modeling and verification approach, and present results of verifying our models against the LSC specification.

Specification: The specification consists of one LSC that contains four subcharts, each with a unique activation condition. Fig. 1 shows the first subchart in the LSC specification. The remaining subcharts are not included in the paper to conserve space. Each subchart contains Kleene stars and plus operators that are translated using the schemes presented in Section 4. Each individual subchart is translated and combined into one large automaton using the subchart translation scheme presented in Fig. 5. The total size of the resulting automaton is 291 states and it describes all possible behaviors of the *Target* and *Initiator* processes.

Modeling: Four different models that implement the behaviors of the BVCI model as described in the individual subcharts of the BVCI specification are created in Promela. Due to the inherent limitations of using Promela as the modeling

Model	States	Memory (MB)	Time(s)
default-ack-request	1.10e+06	82.19	14.5
normal-request	1.1e+06	82.29	14.7
default-response	1568	2.62	0.77
normal-response	1574	2.62	0.90

Table 1
Verification results for Promela models against the BVCI protocol LSC specification.

language, the clock signal is abstracted and replaced with a synchronous message that is exchanged between the *Initiator* and *Target* processes.

Verification: To verify the models against the specification, each model is combined with the automaton translated from the BVCI LSC specification using SPIN. A synchronous composition of the two is then checked for accepting cycles using the built in Double Depth First Search (DDFS) algorithm of SPIN. Results are shown in Table 1. For each model, we list the number of states that were explored to completely verify the model against the entire BVCI specification along with the memory and time resources that were utilized. The results presented here are for models that did not contain any errors. In a separate verification exercise, safety and liveness errors were introduced in the models and verified against the specification. Each error introduced in the model was successfully discovered by the model checker. Since past techniques for LSC verification do not incorporate Kleene star and subchart translations, a comparison is not possible.

6 Conclusions

We have shown how additional constructs of LSCs such as subcharts, Kleene stars, and hierarchical charts can be translated to a negative automaton. We have also presented a case study of using our translation technique to create an automaton from the BVCI protocol and perform verification of Promela models against the resulting specification, which has not been done before. Our results and experiences indicate that using the LSC language as a specification language is extremely useful for writing and developing specifications that can be used during formal verification. Additionally, their use as a modeling language further strengthens their applicability in the initial stages of the protocol development process.

For future work in this area we are developing a technique that allows us to create individual negative automata for each process/instance described in the LSC to perform modular verification of a system and reduce the verification state space. We are also working towards incorporating the current approach in a tool chain for protocol development/verification.

References

[1] Alur, R. and M. Yannakakis, *Model checking of message sequence charts*, in: *Proc. of the 10th Int. Con. on Concurrency Theory (CONCUR99)* (1999), pp. 114–129.

[2] Bohn, J., W. Damm, J. Klose, A. Moik and H. Wittke, *Modeling and validating train system applications using statemate and live sequence charts*, Proc. of the 6th Biennial World Conf. on Integrated Design and Process Technology (IDPT02) (2002).

[3] Bontemps, Y., P. Heymans and 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).

[4] Brill, M., W. Damm, J. Klose, B. Westphal and 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** (2004), pp. 374–399.

[5] Bunker, A., G. Gopalakrishnan and K. Slind, *Live sequence charts applied to hardware requirements specification and verification*, Int. Journal on Software Tools for Technology Transfer (STTT) **7** (2005), pp. 341–350.

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

[7] Harel, D., H. Kugler and A. Pnueli, *Synthesis revisited: Generating statechart models from scenario-based requirements*, Formal Methods in Software and Systems Modeling (2005), pp. 309–324.

[8] Holzmann, G., "The Spin Model Checker: Primer and Reference Manual," Addison-Wesley Professional, 2004.

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

[10] Klose, J., "Live sequence charts: A graphical formalism for the specification of communication behavior," Ph.D. thesis, Fachbereich Informatik, Carl Von Ossietzky University (2003).

[11] Kugler, H., D. Harel, A. Pnueli, Y. Lu and Y. Bontemps, *Temporal logic for scenario-based specifications*, Proc. of the 11th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS05) **3440** (2005), pp. 445–460.

[12] Kumar, R. and E. Mercer, *Improved live sequence chart to automata translation for verification*, in: *Proc. of the 7th Int. Worksop on Graph Transformation and Visual Modeling Techniques (GT-VMT08), Budapest, Hungary*, 2008, pp. 51 – 64.

[13] Kumar, R., E. Mercer and A. Bunker, *Improving translation of live sequence charts to temporal logic*, in: *Proc. of the 7th Int. Conf. on Automated Verification of Critical Systems (AVoCS07)*, 2007, pp. 183 – 197.

[14] Mauw, S. and M. A. Reniers, *High-level message sequence charts*, in: *Proceedings of the Eighth SDL Forum (SDL97)*, 1997, pp. 291–306.

[15] Mauw, S. and M. A. Reniers, *Operational Semantics for MSC'96*, Computer Networks (Amsterdam, Netherlands: 1999) **31** (1999), pp. 1785–1799.

[16] Sun, J. and J. S. Dong, *Model checking live sequence charts*, in: *ICECCS05: Proc. of the 10th IEEE Int. Conf. on Engineering of Complex Computer Systems (ICECCS05)* (2005), pp. 529–538.