Singapore Management University

Institutional Knowledge at Singapore Management University

Research Collection School Of Computing and Information Systems | School of Computing and Information Systems

7-2005

Synthesis of distributed processes from scenario-based specifications

Jun SUN
Singapore Management University, junsun@smu.edu.sg

Jin Song DONG

Follow this and additional works at: https://ink.library.smu.edu.sg/sis_research

Part of the Programming Languages and Compilers Commons, and the Software Engineering Commons

Citation
SUN, Jun and DONG, Jin Song. Synthesis of distributed processes from scenario-based specifications. (2005). *Proceedings of 2005 International Symposium of Formal Methods Europe, Newcastle, UK, July 18-22*. 415-431.
Available at: https://ink.library.smu.edu.sg/sis_research/5056

This Conference Proceeding Article is brought to you for free and open access by the School of Computing and Information Systems at Institutional Knowledge at Singapore Management University. It has been accepted for inclusion in Research Collection School Of Computing and Information Systems by an authorized administrator of Institutional Knowledge at Singapore Management University. For more information, please email cherylds@smu.edu.sg.

Synthesis of Distributed Processes from Scenario-Based Specifications

Jun Sun and Jin Song Dong

School of Computing, National University of Singapore
{sunj, dongjs}@comp.nus.eud.sg

Abstract. Given a set of sequence diagrams, the problem of synthesis is of deciding whether there exists a satisfying object system and if so, synthesize one automatically. It is crucial in the development of complex systems, since sequence diagrams serve as the manifestation of use cases and if synthesizable they could lead directly to implementation. It is even more interesting (and harder) if the synthesized object system is distributed. In this paper, we propose a systematic way of synthesizing distributed processes from Live Sequence Charts. The basic idea is to first construct a CSP specification from the LSC specification, and then use CSP algebraic laws to group the behaviors of each object effectively. The key point is that the behaviors of each object can be decided locally without constructing the global state machine.

Keywords: LSC, CSP, Synthesis.

1 Introduction

Sequence diagrams have been a popular means of specifying scenarios of reactive systems for decades. They have found their ways into many methodologies, e.g. Sequence Diagrams in Unified Modelling Languages (UML [12]), Messages Sequence Charts (MSCs) in Specification and Description Language (SDL) [18]. They are used in the early stage of system development to describe possible communication scenarios. Given a set of sequence diagrams, the problem of synthesis is of deciding whether there exists a satisfying object system and if so, synthesize one automatically. The problem is crucial in the development of complex systems, as sequence diagrams serve as the manifestation of use cases and if synthesizable they could lead directly to implementation. The problem has been long recognized as a hard problem and tackled by many researchers [2, 1, 21]. The conclusion is that for reactive distributed systems, synthesizing a distributed object system with precisely the set of behaviors is in general impossible. Detailed discussions on why distributed systems are hard to synthesize and why unspecified behaviors are unavoidable can be found in [24] and [1] respectively.

Live Sequence Charts (LSCs) are proposed by Damm and Harel [8]. They are rapidly recognized as a rather rich and useful extension of MSCs. A rich set of constructs are provided for specifying not only possible behaviors, but also mandatory behaviors. For instance, a universal chart, possibly preceded with a pre-chart, specifies mandatory behaviors globally, i.e. once the system behavior matches its pre-chart, the

J.S. Fitzgerald, I.J. Hayes, and A. Tarlecki (Eds.): FM 2005, LNCS 3582, pp. 415–431, 2005.
© Springer-Verlag Berlin Heidelberg 2005

subsequence behavior must follow the chart. On the level of a chart, events and conditions and locations are also labelled with modalities. LSCs also provides structuring constructs, like sub-charts, branching and iterations, to build scenarios hierarchically. In a nutshell, LSCs provide a far more powerful means for setting requirements for complex system than classic sequence diagrams. Therefore, they serve as the basis of tool supporting analysis of scenarios, for example, the study of the synthesis problem.

The synthesis problem of LSCs is discussed by Harel and Kugler in [14], in which they tackled the problem by defining the notion of consistency between LSCs. Their approach starts with constructing a **global system automata** and decompose it by different means (refer to [14] for details). Their approach suffers from the state explosion problem due to the construction of the **global system automata**, which is often of huge size because of the distributed nature of LSCs and the underlying weak partial ordering semantics. In this paper, we present a systematic way of synthesizing distributed processes directly from LSCs. The basic idea is to first construct a Communicating Sequential Process (CSP [17]) specification from the LSC specification, and then use CSP algebraic laws to group the behaviors of each object effectively. The key point is that the behaviors of each object can be decided locally without constructing the global state machine. In our previous work [27], we explored the semantic-based equivalence relations between CSP and LSCs. We prove that we may capture the semantics of LSC specifications using CSP. The practical implication is that CSP supporting tools like FDR [10] can be reused to validate LSC specifications. The construction of CSP specifications in this work, however, is different because our aim is to synthesize refinements of consistent LSC specifications. Only distributed processes that are not only consistent with the LSC specification but also regular (so that they lead to finite state machine implementations) and minimally restrictive (if possible) are interested. Our work in [27] can be viewed as a necessary precedence of this work. Our approach is experimented with an automated tool developed using JAVA and XML.

We remark that the same result can be derived using Büchi Automata [6] with a painfully complicated procedure. In [3], Bontemps and Heymans use Büchi automata to define the language expressed by a set of LSCs. They claim that standard algorithm for automata can be used to check consistency and refinement and etc. As one of the future works, they mentioned the synthesis of state-based implementations from LSCs. However, as Büchi automata are low-level and not structured, flattening high-level LSCs into automata suffers from the state explosion problem. Whereas CSP provides a rich set of compositional constructs. Therefore, our work preserves the structure of the LSC specification and avoids constructing the global state machine both at the chart level or globally. In [4], Bontemps and Schobbens and Löding discussed the synthesis problem for a small subset of LSCs (LSCs without conditions, structuring constructs, modalities on locations and messages). They proposed a game-based semantics for LSCs, which leads to the notion of consistency between their LSCs. However, their discussion on the problem of synthesis is limited to a single universal chart. In our approach, almost all LSC constructs are supported except timing constructs, which we leave to the future works. In addition, there is the work described in [19], which synthesizes a timed Büchi Automata from a single chart only. What makes our goal both harder and more interesting is in the treatment of a set of charts, not just a single one. As far as the limited

case of classical MSC goes, there have been quite some works on formalizing and then synthesizing from them. This includes the works by Alur mentioned earlier and others, evidenced in [20, 21, 22, 16].

The rest of the paper is organized as follows. Section 2 introduces LSCs and CSP. Section 3 presents our approach to synthesize distributed processes from a set of LSC universal charts. Section 4 discusses relevant issues of the synthesis, i.e. how to handle modalities on locations. Section 5 concludes the paper with possible future works.

2 Background

2.1 Live Sequence Charts

MSCs are widely used to describe scenarios of interaction between processes or objects. However, MSCs suffer from the rather weak partial-order semantics that makes it incapable of capturing many kinds of behavioral requirements. LSCs are introduced in [8] to overcome the shortcomings of MSCs by adding liveness or universality, i.e. something desired must be observed.

There are two kinds of charts in LSCs. Existential charts are mainly used to describe possible scenarios of a system in the early stage of system development, i.e. the same role played by classic MSCs. In later stage, knowledge becomes available about when a system run has progressed far enough for a specific usage of the system to become relevant. Universal charts are then used to specify behaviors that should always be exhibited. In this work, we assume that an LSC specification consists of a set of universal charts, and existential charts are used to specify test cases. A universal chart may be preceded by a pre-chart, which serves as the activation condition for executing the main chart. Whenever a communication sequence matches a pre-chart, the system must proceed as specified by the main chart. Due to pre-charts, a system run may activate a universal chart more than once and some of the activation might overlap [23].

Each chart is associated with a set of visible events. Only the set of visible events are constrained by the chart. A chart typically consists of multiple instances, which are represented as vertical lines graphically. Along with each line, there are a finite number of locations. A location carries the temperature annotation for progress within an instance. A location may be labelled as either cold or hot. A hot location means that the system has to move beyond. Whereas the system may stay at a cold location forever. Similarly, messages and conditions are also labelled. A hot message must be received, whereas a cold one may get lost. A hot condition must be met, whereas a cold condition terminates the chart if it is evaluated to false.

Example 1. We introduce a mobile phone system as a running example to explain and illustrate the main ideas and results. This example is partially inspired by the phone system specification presented in [7]. The system consists of six participating objects, a $user$, the $cover$, the $display$, the $speaker$, the $chip$ and the environment where the incoming calls are from. Due to the page limit, we only introduce a self-containing set of scenarios. Scenario **OpenCover** illustrates the interaction between the objects when the $user$ opens the $cover$, i.e. the $chip$ is notified that the $cover$ is opened, it then requests the $display$ to display the menu. The $display$ then carries out a local action

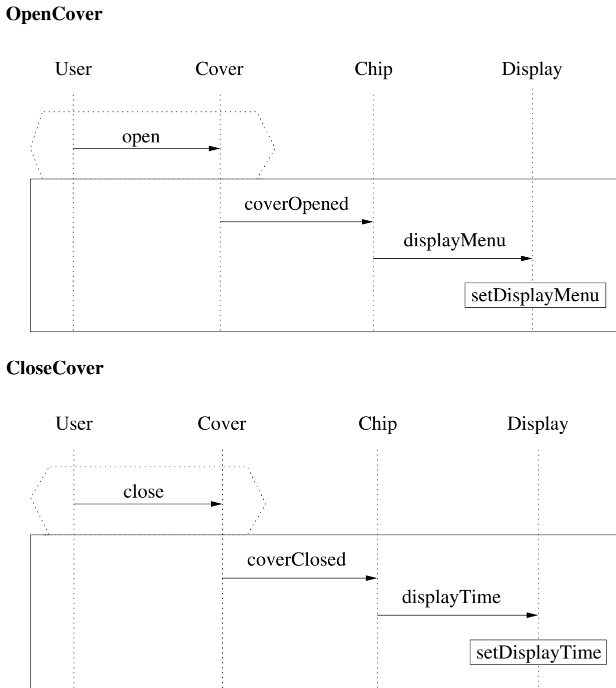
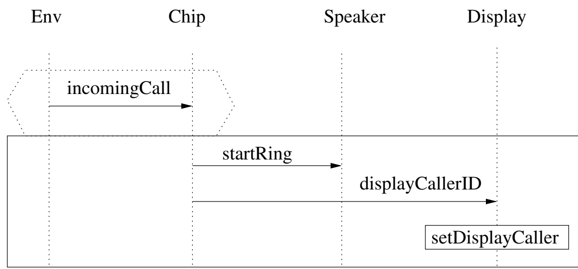


Fig. 1. Mobile Phone System Scenario: OpenCover, CloseCover

$setDisplayMenu$ to initialize the menu screen. The upper chart in Figure 1 illustrates the scenario. Figure 1, 2 illustrates the scenarios where the user closes the cover, an incoming call arrives and the user picks up the phone and talk. These scenarios are self-explanatory. Note that all vertical lines in the charts are dotted, which means that all locations along the lines are cold and, therefore, the system may pause at any point of execution forever. This is possible because unexpected events like the battery runs out or the system breaks down may occur at any time. The set of visible events for each chart are exactly those appeared in the diagram except the scenario **Talk**, which includes a forbidden event $close$. We remark that the message from the user to the cover $close$ is forbidden in the scenario **Talk**, i.e. in order to carry out the scenario successfully, the user should not close the cover before the scenario completes. Figure 3 illustrates the typical usage of the phone. Note that implicit assumptions are captured by hot locations, for example an incoming call will eventually trigger the ring, the user will eventually pick up the call and hand up the call and etc.

LSCs also support advanced MSC features like co-region, hierarchy and etc. Symbolic instances and messages are adopted to group scenarios effectively. For a detailed introduction on a complete list of features of LSCs, refer to [15]. LSCs are far more expressive than MSCs, which makes them capable of expressing complicated scenario-based requirements. However, we remark that the ability to specify hot and cold messages, i.e. whether a message is required to be received or may get lost, is redundant

Receive



Talk

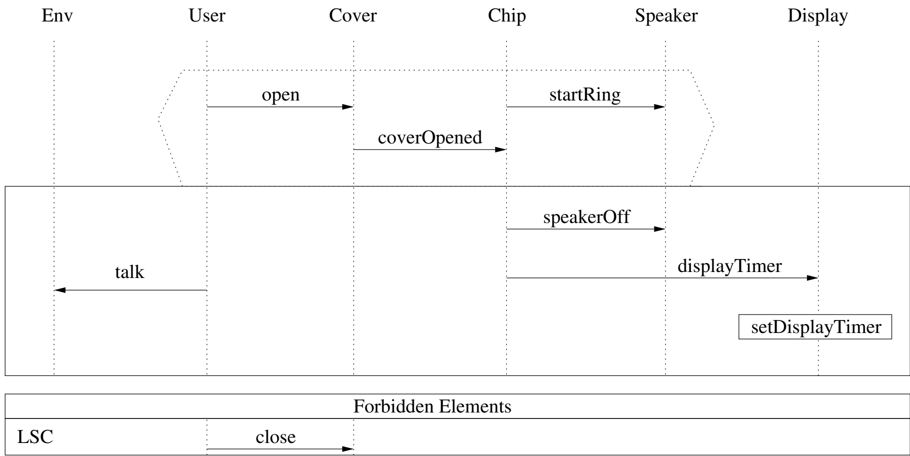


Fig. 2. Mobile Phone System Scenario: Receive, Talk

Phone

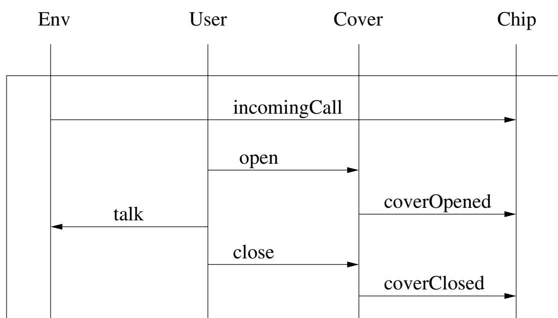


Fig. 3. Mobile Phone System Scenario: Phone

because of the facility for describing hot and cold locations. Essentially, the temperature of the locations takes precedence over the temperature of messages, so whether or not the message is received is determined entirely by the temperature of the message input. This questionable feature of LSCs is recognized by Harel and Marelly who list the possible cases and conclude that the temperature of the message has no semantical meaning [15]. Thus, in the following discussion, the temperature of all messages is discarded.

2.2 Communicating Sequential Process

Hoare's CSP [17, 25] is a formal specification language where processes proceed from one state to another by engaging in events. Processes may be composed by operators which require synchronization on events, i.e. each component must be willing to participate in a given event before the whole system makes the transition. A CSP process is defined by process expressions. Let $\mathcal{P}$ denote all possible CSP processes. The relevant syntactic class of process expression is defined as:

$$\begin{aligned}\mathcal{P} ::= \;& RUN_{\Sigma} \mid \textsc{Stop} \mid \textsc{Skip} \mid \mathcal{P}_1 \sqcap \mathcal{P}_2 \mid \mathcal{P}_1 \Box \mathcal{P}_2 \mid \mathcal{P}_1;\ \mathcal{P}_2 \mid \mathcal{P}_1 \mid\mid\mid \mathcal{P}_2 \mid \\ & \mathcal{P}_1 \;{}_X\|_Y\; \mathcal{P}_2 \mid \mathcal{P}_1 \;{}_X\|_Y\; \mathcal{P}_2 \mid \mathcal{P}_1 \nabla_e \mathcal{P}_2 \mid \cdots\end{aligned}$$

CSP defines a rich set of operators to create processes. RUN_{Σ} is a process always willing to engage any event in Σ. STOP denotes a process that deadlocks and does nothing. A process that terminates is written as SKIP. A process $e \rightarrow \mathcal{P}$ is initially willing to engage in event e and behaves as $\mathcal{P}$ afterward. CSP allows a hierarchical description of a system by offering various operators to compose processes. The sequential composition, $\mathcal{P}_1;\ \mathcal{P}_2$, behaves as $\mathcal{P}_1$ until its termination and then behaves as $\mathcal{P}_2$. A choice between two processes is denoted as $\mathcal{P}_1 \mid \mathcal{P}_2$. The choice is made either internally ($\mathcal{P}_1 \sqcap \mathcal{P}_2$) or externally ($\mathcal{P}_1 \Box \mathcal{P}_2$). Often, choices are guarded by prefixing or conditionals. A choice that depends on the truth value of a boolean expression b is written as $\mathcal{P}_1 \not\leftarrow b \not\rightarrow \mathcal{P}_2$. If b is true, this process proceeds as $\mathcal{P}_1$, otherwise $\mathcal{P}_2$. Parallel composition of two processes is denoted as $\mathcal{P}_1 \parallel \mathcal{P}_2$, where common events are synchronized. If X is an empty set, the two processes interleaves, denoted as $\mathcal{P}_1 \mid\mid\mid \mathcal{P}_2$. The generalized form of synchronization is denoted as $\mathcal{P}_1 \;{}_X\|_Y\; \mathcal{P}_2$, the alphabetized parallel composition where common events in X and Y are synchronized. $\|_{k=1}^{n}(\mathcal{P}_k, \Sigma_k)$ is a replicated alphabetized parallel denoting parallel composition of n processes, where each process $\mathcal{P}_k$ synchronizes with the rest of the system on events in Σ_k. $\mathcal{P}_1 \nabla_e \mathcal{P}_2$ behaves as $\mathcal{P}_1$ until event e is engaged and then $\mathcal{P}_2$ takes control.

Three mathematical models for CSP are defined. In the traces model, a process is represented by the set of finite sequences of communications it can perform, denoted as $traces(\mathcal{P})$. In the stable failures model, a process is represented by its traces and also by its failures. A failure is a pair (t, Σ), where t is a finite trace of the process and Σ is a set of events it can refuse after t (refusal). The set of P's failures is denoted as $failures(\mathcal{P})$. In the failures/divergences model [5], a process is represented by its failures, together with its divergences. A divergence is a finite trace during or after which the process can perform an infinite sequence of consecutive internal actions. Failure/divergence model and stable failure model make no difference for divergence-free systems. A detailed discussion on the three semantics models can be found in [25]. The well-established

failure semantics is used to establish equivalence relations between processes by appeal to algebraic laws of CSP. We quote the relevant laws below. The proof of each law can be found in either [17] or [25].

$$\mathcal{P} \,|[\, \Sigma \,]|\, RUN_{\Sigma} = \mathcal{P} \quad \text{[L1]}$$
$$\mathcal{P} \parallel \textsc{Stop} = \textsc{Stop} \quad \text{[L2]}$$
$$\mathcal{P} \parallel \mathcal{P} = \mathcal{P} \quad \text{[L3]}$$
$$\mathcal{P}_1 \; {}_X\|_Y \; \mathcal{P}_2 = \mathcal{P}_2 \; {}_Y\|_X \; \mathcal{P}_1 \quad \text{[L4]}$$
$$(\mathcal{P}_1 \; {}_X\|_Y \; \mathcal{P}_2) \; {}_{X\cup Y}\|_Z \; \mathcal{P}_3 = \mathcal{P}_1 \; {}_X\|_{Y\cup Z} \; (\mathcal{P}_2 \; {}_Y\|_Z \; \mathcal{P}_3) \quad \text{[L5]}$$

The following laws are derived. Law [L6] is a directly consequence of law [L4] and [L5]. Law [L7] is the generalized form of law [L6].

$$(\mathcal{P}_1 \; {}_X\|_Y \; \mathcal{P}_2) \; {}_{X\cup Y}\|_{Z\cup W} \; (\mathcal{P}_3 \; {}_Z\|_W \; \mathcal{P}_4) = (\mathcal{P}_1 \; {}_X\|_Z \; \mathcal{P}_3) \; {}_{X\cup Z}\|_{Y\cup W} \; (\mathcal{P}_2 \; {}_Y\|_W \; \mathcal{P}_4) \quad \text{[L6]}$$
$$\Big\|_{i=1}^{m} \big(\Big\|_{j=1}^{n} (\mathcal{P}_i^j, \Sigma_i^j), \bigcup_j \Sigma_i^j\big) = \Big\|_{j=1}^{n} \big(\Big\|_{i=1}^{m} (\mathcal{P}_i^j, \Sigma_i^j), \bigcup_i \Sigma_i^j\big) \quad \text{[L7]}$$

3 Synthesizing Distributed Processes

Our discussion in this section assumes that the LSC specification is well-formed and consistent, i.e. the weak event relation is acyclic, existential charts trace-refine the universal charts and etc. Additional assumptions are discussed in the following. We assume that all locations are cold and all conditions are distributed. The former is due to the lack of "liveness" in the original CSP semantics. This problem is addressed in Section 4. The latter gets rid of shared condition, which we think is a problematic feature of LSCs. In LSCs, a condition is a boolean expression over the visible variables of the chart. Therefore, some form of global variables is presupposed. This doesn't match the reality of distributed system. Indeed, objects in distributed systems have their own state space (local variables) and all communication between objects would be via messages. Therefore, we are only interested in local conditions in this work. However, shared condition can be (partially) supported by rewriting it to a set of distributed condition with additional proper synchronization. Without loss of generality, we also assume that no co-region is allowed and all messages are synchronized. There is nothing interesting about co-region except it complicates the presentation of the synthesis. Asynchronous message passing is supported by explicitly modelling the behavior of the buffers, e.g. FIFO. A consequence of this assumption is that a message loss is captured by an infinitely long delay of the forwarding by the buffer instead of a traditional $lost\ message$ symbol.

The principles of the synthesis are that, the synthesized processes should be minimally restrictive (if possible) so that further refinement is possible, the global state machine should never be constructed so that state explosion is avoided, and above all, the synthesized processes should be consistent with the LSC specification. The basic idea of our approach is to first construct a CSP specification from the LSC specification (a refinement), and then use CSP algebraic laws to group the behaviors of each object effectively. The key point of our synthesis is that the behaviors of each object can be determined locally and, therefore, the global state machine is never constructed. In the following, we present the synthesis in a bottom-up fashion using synthesis rules (**SR**).

The most primitive building blocks of LSCs are locations. Along an instance in a chart, there are a finite number of locations. A location contains exactly one event and an optional condition. Let $\mathcal{S}$ be an LSC specification. Let c, i be a chart and a participating object (instance) in $\mathcal{S}$ respectively. Let $Location, Condition, Event$ be all locations, condition and events respectively. Let $cond : Location \rightarrow Condition$ be the condition observer. Let $event : Location \rightarrow Event$ be the event observer. We denote the process synthesized for the location l on instance i in the main chart of chart c as $MainLoca_c^i(l)$. Let $MainLoca_c^i(l+1)$ be the process synthesized for the next location.

- **SR1**: The condition labelled with location l is cold and location l is not the last. If the condition labelled with l evaluates to true, the system engages the event and proceeds to the next location, otherwise, its engages a special event α_c to signal all other instances in the chart before termination. Processes for all other instances in the chart are interrupted by α_c and terminate so that the chart terminates.

$$MainLoca_c^i(l) \widehat{=} (event(l) \rightarrow MainLoca_c^i(l+1)) \not\leftarrow cond(l) \not\rightarrow (\alpha_c \rightarrow \textsc{Skip})$$

- **SR2**: The condition is cold and the location is the last. After engaging the event, a special event γ_c is synchronized by all instances in the chart before any of them terminates.

$$MainLoca_c^i(l) \widehat{=} (event(l) \rightarrow \gamma_c \rightarrow \textsc{Skip}) \not\leftarrow cond(l) \not\rightarrow (\alpha_c \rightarrow \textsc{Skip})$$

- **SR3**: The condition is hot and the location is not the last. A special event β_c is engaged if the hot condition is violated so that all other instances in the chart are signaled and deadlock.

$$MainLoca_c^i(l) \widehat{=} (event(l) \rightarrow MainLoca_c^i(l+1)) \not\leftarrow cond(l) \not\rightarrow (\beta_c \rightarrow \textsc{Stop})$$

- **SR4**: The condition is hot and the location is the last.

$$MainLoca_c^i(l) \widehat{=} (event(l) \rightarrow \gamma_c \rightarrow \textsc{Skip}) \not\leftarrow cond(l) \not\rightarrow (\beta_c \rightarrow \textsc{Stop})$$

Each chart is associated with a set of visible events. Let Σ_c be the set of visible events of chart c. Let Σ_c^i be the set of events associated with Instance i in chart c, including forbidden events. Special events are added to Σ_c^i to carry out the synthesis systematically. The number of special events is bounded by the number of charts if we are only interested in regular implementations (discussed later). In particular, we associate each chart with three special events, $\alpha_c, \beta_c, \gamma_c$. Event α_c is engaged only when a cold condition is violated, either in the pre-chart or the main chart. Event γ_c is used to synchronize the entering or exiting of a chart or a sub-chart among all participating instances. For example, in the above construction, a γ_c event is engaged when the last location has been traversed. Event β_c is engaged only when a hot condition is violated so as to force the system to fail. This reflects the semantics of hot conditions. However, this is slightly problematic as the intention of hot conditions is to make sure they are never violated in the scenario. A hot condition is violated either because there is inconsistency in the LSC specification, i.e. wrong implementation of the local action and etc., or the system is insufficiently specified. A model checker, e.g. FDR, would help refine LSC specifications step by step so that all hot condition holds all the time [27].

A location could be a structuring construct, e.g. a sub-chart or a branching. We remark that all LSC structuring constructs have their exact images in CSP, e.g. choice in CSP for branching, process reference for sub-charts and etc. This is a clear advantage why CSP is better than unstructured automata for our discussion.

Similarly, we may synthesize the process for a location l in the pre-chart. We denote the process synthesized for the location l on instance i in the pre-chart of chart c as $PreLoca_c^i(l)$. An instance not in the pre-chart is treated as if it is in the pre-chart with one empty location.

- **SR5**: Location l is neither the first location nor the last. If the condition evaluates to false, then the process signals all other instances in the chart and terminates. Otherwise, if the expected event is engaged, the process proceeds to the next location, else, the process engages the unexpected event and puts no further constraints on the system ([L1]). Note that we do not distinguish hot or cold condition in pre-charts as hot conditions have no semantical meaning in pre-charts.

$$\begin{aligned} PreLoca_c^i(l) \mathrel{\widehat=} &((event(l) \rightarrow PreLoca_c^i(l+1)) \\ &\quad \Box\ (\Box\, e : \Sigma_c^i \setminus \{event(l), \alpha_c, \beta_c, \gamma_c\} \rightarrow RUN)) \\ &\not\leftarrow cond(l) \not\rightarrow (\alpha_c \rightarrow \textsc{Skip}) \end{aligned}$$

- **SR6**: The location is not the first location but is the last. After engaging the event, the instance waits for the synchronization for termination and proceeds to the first location of the main chart.

$$\begin{aligned} PreLoca_c^i(l) \mathrel{\widehat=} &((event(l) \rightarrow \gamma_c \rightarrow MainLoca_c^i(0)) \\ &\quad \Box\ (\Box\, e : \Sigma_c^i \setminus \{event(l), \alpha_c, \beta_c, \gamma_c\} \rightarrow RUN)) \\ &\not\leftarrow cond(l) \not\rightarrow (\alpha_c \rightarrow \textsc{Skip}) \end{aligned}$$

- **SR7**: The location is the first but not the last. A new process is forked whenever an expected event is engaged. This way, we allow system runs that may trigger multiple overlapping activation of the same chart. Note that the special events are not synchronized between different activation.

$$\begin{aligned} PreLoca_c^i(0) \mathrel{\widehat=} &((event(0) \rightarrow PreLoca_c^i(1) \,|[\, \Sigma_c^i \setminus \{\alpha_c, \beta_c, \gamma_c\} \,]|\, PreLoca_c^i(0)) \\ &\quad \Box\ (\Box\, e : \Sigma_c^i \setminus \{event(0), \alpha_c, \beta_c, \gamma_c\} \rightarrow RUN)) \\ &\not\leftarrow cond(0) \not\rightarrow (\alpha_c \rightarrow PreLoca_c^i(0)) \end{aligned}$$

- **SR8**: The location is the only location of the instance in the pre-chart.

$$\begin{aligned} PreLoca_c^i(0) \mathrel{\widehat=} &((event(0) \rightarrow \\ &\quad (\gamma_c \rightarrow MainLoca_c^i(0)) \,|[\, \Sigma_c^i \setminus \{\alpha_c, \beta_c, \gamma_c\} \,]|\, PreLoca_c^i(0)) \\ &\quad \Box\ (\Box\, e : \Sigma_c^i \setminus \{event(0), \alpha_c, \beta_c, \gamma_c\} \rightarrow RUN)) \\ &\not\leftarrow cond(l) \not\rightarrow (\alpha_c \rightarrow PreLoca_c^i(0)) \end{aligned}$$

- **SR9**: The chart is not preceded with a pre-chart.

$$PreLoca_c^i(0) \mathrel{\widehat=} MainLoca_c^i(0)$$

Whenever a chart is activated by a system run, the subsequence behavior of the system is constrained by both the process and the newly forked process and, therefore, remains valid. However, the process $PreLoca_c^i(0)$ allows, in general, irregular languages that cannot be realized by finite state machines. A similar problem is recognized by Harel and Kugler [14]. We may synthesize systems with possible overlapping activation of the same chart using the above set of rules. Nevertheless, in most cases, only regular processes which lead to finite state implementations are interested. If we assume that activation of the same chart never overlaps, i.e. the chart is not re-activated till its completion, we may augment **SR1-2, SR4-8** as the following so that a chart can be re-activated only after its completion. The same assumption is made by Harel and Kugler in [14].

SR1': $MainLoca_c^i(l) \mathrel{\widehat=} (event(l) \rightarrow MainLoca_c^i(l+1)) \not\leftarrow cond(l) \not\rightarrow (\alpha_c \rightarrow PreLoca_c^i(0))$
SR2': $MainLoca_c^i(l) \mathrel{\widehat=} (event(l) \rightarrow \gamma_c \rightarrow PreLoca_c^i(0)) \not\leftarrow cond(l) \not\rightarrow (\alpha_c \rightarrow PreLoca_c^i(0))$
SR4': $MainLoca_c^i(l) \mathrel{\widehat=} (event(l) \rightarrow \gamma_c \rightarrow PreLoca_c^i(0)) \not\leftarrow cond(l) \not\rightarrow (\beta_c \rightarrow \text{STOP})$
SR5': $PreLoca_c^i(l) \mathrel{\widehat=} ((event(l) \rightarrow PreLoca_c^i(l+1)) \,\Box$
$(\Box\, e : \Sigma_c^i \setminus \{event(l), event(0), \alpha_c, \beta_c, \gamma_c\} \rightarrow PreLoca_c^i(0)))$
$\not\leftarrow cond(l) \not\rightarrow (\alpha_c \rightarrow PreLoca_c^i(0))$
SR6': $PreLoca_c^i(l) \mathrel{\widehat=} ((event(l) \rightarrow \gamma_c \rightarrow MainLoca_c^i(0)) \,\Box$
$(\Box\, e : \Sigma_c^i \setminus \{event(l), event(0), \alpha_c, \beta_c, \gamma_c\} \rightarrow PreLoca_c^i(0)))$
$\not\leftarrow cond(l) \not\rightarrow (\alpha_c \rightarrow PreLoca_c^i(0))$
SR7': $PreLoca_c^i(0) \mathrel{\widehat=} ((event(0) \rightarrow PreLoca_c^i(1)) \,\Box$
$(\Box\, e : \Sigma_c^i \setminus \{event(0), \alpha_c, \beta_c, \gamma_c\} \rightarrow PreLoca_c^i(0)))$
$\not\leftarrow cond(l) \not\rightarrow (\alpha_c \rightarrow PreLoca_c^i(0))$
SR8': $PreLoca_c^i(0) \mathrel{\widehat=} ((event(0) \rightarrow \gamma_c \rightarrow MainLoca_c^i(0)) \,\Box$
$(\Box\, e : \Sigma_c^i \setminus \{event(0), \alpha_c, \beta_c, \gamma_c\} \rightarrow PreLoca_c^i(0)))$
$\not\leftarrow cond(l) \not\rightarrow (\alpha_c \rightarrow PreLoca_c^i(0))$

Rule **SR1-2,4** are augmented so that the process proceeds to the first location after completing the last location or whenever a cold condition is violated. Rule **SR5-8** are augmented so that the initial event which may activate a chart ($event(0)$) is not engaged before the chart completes. As a result, no new processes need to be forked under our assumption. For simplicity, the subsequent discussion assumes that there is no overlapping activation of the same chart. The process synthesized for instance i in chart c is denoted as $Instance_c^i$.

– **SR10**: The process terminates whenever a cold condition is violated in the chart, and deadlocks whenever a hot condition is violated. Both are captured using interrupt operators.

$$Instance_c^i \mathrel{\widehat=} (PreLoca_c^i(0) \bigtriangledown_{\alpha_c} Instance_c^i) \bigtriangledown_{\beta_c} \text{STOP}$$

Each chart consists of a finite number of interacting instances. Let $Chart_c$ be the process for chart c.

– **SR11**: The process is an alphabetized parallel of the processes of all instances in the chart. Note that in case a hot condition is violated, the process deadlocks and, therefore, the system deadlocks (**L2**). In case a cold condition is violated, the process restores to its initial state.

$$Chart_c \mathrel{\widehat{=}} \Big|\Big|_i (Instance_c^i, \Sigma_c^i)$$

An LSC specification consists of a finite number of universal charts, each constraining a set of visible events. Let $\mathcal{I}$ be the process synthesized from the LSC specification.

– **SR12**: $\mathcal{I} \mathrel{\widehat{=}} \Big|\Big|_c (Chart_c, \Sigma_c)$

We claim that $\mathcal{I}$ is an implementation of $\mathcal{S}$. From the construction of $Chart_c$, it is clear that only behaviors satisfying the chart are allowed. Therefore, $\mathcal{I}$ only allows behaviors that satisfies all the charts (because of the parallel composition). Moreover, $Chart_c$ only constraints its visible events (as it is alphabetized) and, therefore, other events are free to occur. We skip the case-by-case proof in this paper. The main result of our work is that we may group the behaviors of an object in the system effectively by transforming $\mathcal{I}$ using CSP algebraic laws, in particular, the distributivity law of alphabetized parallel composition.

$$\mathcal{I} \mathrel{\widehat{=}} \Big|\Big|_c (Chart_c, \Sigma_c) \mathrel{\widehat{=}} \Big|\Big|_c (\Big|\Big|_i (Instance_c^i, \Sigma_c^i), \Sigma_c) \qquad \textbf{[SR11,12]}$$

$$\mathrel{\widehat{=}} \Big|\Big|_i (\underline{\Big|\Big|_c (Instance_c^i, \Sigma_c^i)}, \bigcup_i \Sigma_c^i) \qquad \textbf{[L7]}$$

We remark that the underlying portion of the process is the behavior of an object in isolation, and $\bigcup_i \Sigma_c^i$ is its alphabet with a number of special events. Thus, the behaviors of each objects can be determined locally without ever constructing the global state machine. Each object is composed with the rest of the system by alphabetized parallel composition. There is a subtle difference between alphabetized parallel composition and traditional common event synchronization between state machines. For the former, an event in the alphabet but not in the process indicates a forbidden event. Whereas the alphabet of a state machine always contains exactly the set of events in the state machine. Other than that, the process of each object is realized by traditional finite state machines straightforwardly.

Example 2. We show the synthesized processes for the lower chart in Figure 2 (as it is the most complicated one) in detail. For the **talk** scenario,

$$\begin{aligned}
Instance_{Talk}^{Env} &\mathrel{\widehat{=}} (\gamma_{Talk} \rightarrow talk \rightarrow \gamma_{Talk} \rightarrow Instance_{Talk}^{Env}) \,\Box\, (talk \rightarrow Instance_{Talk}^{Env}) \\
Instance_{Talk}^{User} &\mathrel{\widehat{=}} (open \rightarrow \gamma_{Talk} \rightarrow talk \rightarrow \gamma_{Talk} \rightarrow Instance_{Talk}^{User}) \\
&\quad \Box\, (talk \rightarrow Instance_{Talk}^{User}) \,\Box\, (close \rightarrow Instance_{Talk}^{User}) \\
Instance_{Talk}^{Cover} &\mathrel{\widehat{=}} (open \rightarrow coverOpened \rightarrow \gamma_{Talk} \rightarrow \gamma_{Talk} \rightarrow Instance_{Talk}^{Cover}) \\
&\quad \Box\, (coverOpened \rightarrow Instance_{Talk}^{Cover}) \,\Box\, (close \rightarrow Instance_{Talk}^{Cover}) \\
Instance_{Talk}^{Chip} &\mathrel{\widehat{=}} (startRing \rightarrow coverOpened \rightarrow \gamma_{Talk} \rightarrow speakerOff \rightarrow \\
&\qquad displayTimer \rightarrow \gamma_{talk} \rightarrow Instance_{Talk}^{Chip}) \\
&\quad \Box\, (coverOpened \rightarrow Instance_{Talk}^{Chip}) \,\Box\, (speakerOff \rightarrow Instance_{Talk}^{Chip}) \\
&\quad \Box\, (displayTimer \rightarrow Instance_{Talk}^{Chip}) \\
Instance_{Talk}^{Speaker} &\mathrel{\widehat{=}} (startRing \rightarrow \gamma_{Talk} \rightarrow speakerOff \rightarrow \gamma_{Talk} \rightarrow Instance_{Talk}^{Speaker}) \\
&\quad \Box\, (speakerOff \rightarrow Instance_{Talk}^{Speaker}) \\
Instance_{Talk}^{Display} &\mathrel{\widehat{=}} (\gamma_{Talk} \rightarrow displayTimer \rightarrow setDisplayTimer \rightarrow \\
&\qquad \gamma_{Talk} \rightarrow Instance_{Talk}^{Display}) \\
&\quad \Box\, (displayTimer \rightarrow Instance_{Talk}^{Display}) \\
&\quad \Box\, (setDisplayTimer \rightarrow Instance_{Talk}^{Display})
\end{aligned}$$

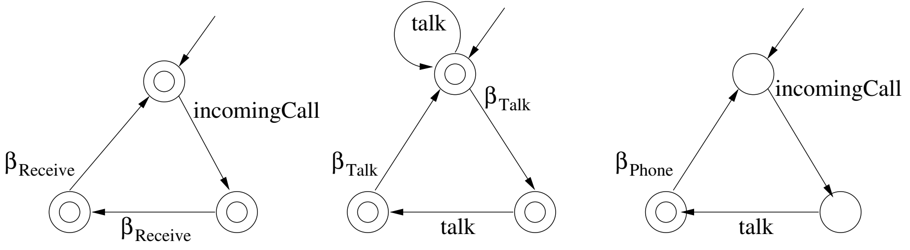


Fig. 4. Finite State Machine Implementation of Env

Before the main chart is activated, all visible events are free to occur (captured by the external choices). Once the instances synchronize the entering of the main chart, only event sequences specified by the main chart are allowed. This example also illustrates how forbidden events are handled. For instance, event $close$ is in the alphabet of $Instance^{User}_{Talk}$ and $Instance^{Cover}_{Talk}$. It can occur before the main chart is activated but not after. Similarly, we may synthesize the processes for the instances in the other charts. The behavior of the same instance is then composed using an alphabetized parallel operator as discussed. The following example shows the behaviors of the Env instance in the system.

$$\begin{array}{ll} Instance^{Env}_{Receive} & \widehat{=} \ incomingCall \rightarrow \gamma_{Receive} \rightarrow \gamma_{Receive} \rightarrow Instance^{Env}_{Receive} \\ Instance^{Env}_{Talk} & \widehat{=} \ (\gamma_{Talk} \rightarrow talk \rightarrow \gamma_{talk} \rightarrow Instance^{Env}_{Talk}) \ \Box \ (talk \rightarrow Instance^{Env}_{Talk}) \\ Instance^{Env}_{Phone} & \widehat{=} \ incomingCall \rightarrow talk \rightarrow \gamma_{phone} \rightarrow Instance^{Env}_{Phone} \\ & \\ Instance^{Env} & \widehat{=} \ Instance^{Env}_{Receive} \parallel Instance^{Env}_{Talk} \parallel Instance^{Env}_{Phone} \end{array}$$

The behaviors of Env in **Receive**, **Talk**, **Phone** are captured by the three processes above. The finite state machine implementation of the Env instance is illustrated in Figure 4. The three state machines are running concurrently, where common events are synchronized. We remark that all the synthesized processes are regular and, therefore, can be implemented by finite state machines.

4 Discussion

In Section 3, we ignore the modalities on locations because CSP lacks the expressiveness to capture liveness, i.e. certain events must be observed in the future. Globally, a system run satisfies an LSC specification only if no instance is stuck at a hot location. In this section, we amend the traditional CSP failure semantics with "signals" to capture liveness. We show that modalities on locations can be captured naturally using signals and the result in Section 3 remains. That is, we show that global behaviors satisfying liveness condition associated with locations can be determined locally.

The name, "signal", is suggested by Davies [9], where signal are used to express broadcast effectively in CSP and they must be observed in the future. In this work, signals are simply events that must be observed in the future. Naturally, events on hot

locations are mapped to signals. In the following discussion, we focus on failure semantics only because there could be nondeterminism in LSCs (therefore trace semantics is insufficient) and there is no hiding operator in LSCs (therefore the synthesized processes are divergence-free). If we use $\widehat{\Sigma}$ to denote the set of all signals, then the set of all events is given by $\widetilde{\Sigma} \mathrel{\widehat{=}} \Sigma \cup \widehat{\Sigma}$. For each event a in Σ, we add a signal $\widehat{a}$. We remark that except they must be engaged eventually, signals play the same role as ordinary events, e.g. synchonizing with signals or events obeying the CSP rules. The set of extended processes is denoted as $\widetilde{\mathcal{P}}$. To ease the discussion (i.e. ensure type consistency), we assume that a process in $\widetilde{\mathcal{P}}$ is uniquely identified by a set of failures.

$$\widetilde{\mathcal{P}} == \mathbb{P}\,\mathbb{P}(\widetilde{\Sigma}^* \times \mathbb{P}\,\widetilde{\Sigma})$$

To ensure the additional constraint caused by signals, we define a filter function to eliminate behaviors from the original CSP failure definitions so that the mature semantic models of CSP are maintained. The filter function $\mathcal{F} : \widetilde{\mathcal{P}} \rightarrow \mathbb{P}\,\widetilde{\mathcal{P}}$ satisfies the following condition:

$$\begin{array}{l} \forall\, p : \widetilde{\mathcal{P}};\ s : \widetilde{\Sigma}^*;\ E : \mathbb{P}\,\widetilde{\Sigma} \bullet \\ \qquad (s, E) \in \mathcal{F}(p) \Leftrightarrow (s, E) \in p \wedge \exists (s, E') : p \bullet \widehat{\Sigma} \subseteq E' \end{array}$$

This axiom insists that any observation that can be extended by engaging a signal must be extended into the future. This way, we augment CSP semantics with a simple fairness condition. Intuitively, it captures the idea that events labelled with a hot location must be engaged. The failures calculation for the compositional CSP constructs remain unchanged and, therefore, the relevant algebraic laws remain valid, including the associativity and symmetry laws for alphabetized parallel ([L4,L5]) and, therefore, law [L6,L7]. The goal of our discussion is to show that the modalities associated with locations can be captured using signals by the processes in a distributed fashion, so that a local process can be implemented by a finite state machine with a set of accepting states. Equivalently, we want to show that in our context the following laws hold.

$$\begin{array}{lll} \mathcal{F}(\widetilde{\mathcal{P}_1}\ {}_X\|_Y\ \widetilde{\mathcal{P}_2}) & = \mathcal{F}(\widetilde{\mathcal{P}_1})\ {}_X\|_Y\ \mathcal{F}(\widetilde{\mathcal{P}_2}) & [\mathsf{L8}] \\ \mathcal{F}(\big\|_{i=1}^{m}(\big\|_{j=1}^{n}(\widetilde{\mathcal{P}}_i^j, \widetilde{\Sigma}_i^j), \bigcup_j \widetilde{\Sigma}_i^j)) & = \big\|_{j=1}^{n}(\mathcal{F}(\big\|_{i=1}^{m}(\widetilde{\mathcal{P}}_i^j, \widetilde{\Sigma}_i^j)), \bigcup_i \widetilde{\Sigma}_i^j) & [\mathsf{L9}] \end{array}$$

Intuitively, if two traces, one for each component, both cannot be extended by engaging a signal, then the composed trace cannot be extended with the signal either. The reserve is not true in general (counter example in Appendix A). However, if a shared signal is always ready to be engaged or refused by both components, then the reverse is also true and, therefore, law [L8, L9] are true. The proof is left to Appendix A. We remark that for consistent LSC specifications, the assumption is safe because graphically a message output event is always connected to a message input events and vice versa due to the absence of *lost message* symbol. The above result is crucial to our work because it guarantees that equipping the global process with liveness conditions is equivalent to equip the liveness conditions locally. Behaviors of each component, therefore, can be decided locally.

Example 3. For example, the CSP process synthesized from *Env* instance in the chart presented in Figure 3 is the following:

$$Instance^{Env}_{Phone} \mathrel{\widehat=} \widehat{incomingCall} \rightarrow \widehat{talk} \rightarrow \gamma_{phone} \rightarrow Instance^{Env}_{Phone}$$

Processes with signals can be implemented as finite state machines equipped with simple fairness conditions, namely, accepting states. A state is accepting if there is no outgoing transition labelled with a signal. For example, in the right-most state machine in Figure 4, only the state after the *talk* event is accepting, indicated by a double-lined circle. A global accepting state is a state where all its components' states are accepting.

5 Conclusion and Future Works

The main contribution of our work is that we present a systematic way of synthesizing distributed processes from LSC specifications. The key point of our method is that the global state machine is never constructed. Therefore, our method can handle system with complicated interactive behaviors. By constructing a CSP specification first and then rewriting it using CSP algebraic laws, we address some of the challenges of such synthesis discussed in [24, 1, 14]. For instance, we prove that the behaviors of each object can be determined without ever constructing the global state machines [14]. We guarantee that no unspecified behaviors are allowed by using only CSP equivalence laws [1]. Moreover, we developed a JAVA application to automatically synthesize CSP expression from LSCs. The tool extends the one reported in [27] with the new way of constructing CSP processes.

There are a couple of possible extensions to our work. First, we may investigate whether our result holds for LSCs with qualitative timing behaviors. Timed CSP [26] seems to be a promising media to carry out the discussion. We may as well transform the synthesized CSP processes to executable models, e.g. SystemC [11], Statechart in Rhapsody [13], so that users may execute the distributed implementations.

Acknowledgements

We thank Steffen Andersen and Dines Bjørner and Steffen Holmslykke for their insightful comments and discussion on early versions of this paper.

References

1. R. Alur, K. Etessami, and M. Yannakakis. Inference of Message Sequence Charts. In *Proc. of the 22nd International Conference on Software Engineering*, pages 304–313. ACM Press, 2000.
2. R. Alur and M. Yannakakis. Model Checking of Message Sequence Charts. In *Proc. of the 10th International Conference on Concurrency Theory*, pages 114–129. Springer-Verlag, 1999.
3. Y. Bontemps and P. Heymans. Turning High-Level Live Sequence Charts into Automata. In *ICSE'02 Workshop: Scenarios and State Machines: Models, Algorithms and Tools*, 2002.
4. Y. Bontemps, P. Schobbens, and C. Löding. Synthesis of Open Reactive Systems from Scenario-Based Specifications. *Fundamenta Informaticae*, 62(2):139–169, July 2004.

5. S. D. Brookes and A. W. Roscoe. An Improved Failures Model for Communicating Processes. In *Proc. of the Pittsburgh seminar on concurrency LNCS 197*, pages 281–305, 1985.
6. J. R. Buchi and L. H. Landweber. Solving Sequential Conditions by Finite State Strategies. *Trans. on American Math. Soc.*, 138:295–311, 1969.
7. D. Harel and R. Marelly. *Play-Engine User's Guide*, 2003.
8. W. Damm and D. Harel. LSCs: Breathing Life into Message Sequence Charts. *Formal Methods in System Design*, 19(1):45–80, 2001.
9. J. Davies. *Specification and Proof in Real-Time CSP*. Cambridge University Press, 1993.
10. Formal System Europe. Failure Divergence Refinement. http://www.fsel.com/, 2003.
11. T. Grotker, S. Liao, G. Martin, and S. Swan. *System Design with SystemC*. Kluwer Academic Publishers, 2002.
12. UML Group. OMG UML v1.5. http://www.uml.org/, June 2002.
13. D. Harel and E. Gery. Executable Object Modeling with Statecharts. *Computer*, 30(7):31–42, 1997.
14. D. Harel and H. Kugler. Synthesizing State-Based Object Systems from LSC Specifications. In *Proc. of CIAA*, volume 2088 of *LNCS*, pages 1–26, 2001.
15. D. Harel and R. Marelly. *Come, Let's Play - Scenario-Based Programming Using LSCs and Play-Engine*. Springer-Verlag, 2003.
16. Ø. Haugen and K. Stølen. STAIRS C Steps to Analyze Interactions with Refinement Semantics. In *Proc. Sixth International Conference on UML (UML'2003)*, volume 2863 of *LNCS*, pages 388–402, 2003.
17. C. A. R. Hoare. *Communicating Sequential Processes*. International Series in Computer Science. Prentice-Hall, 1985.
18. ITU. *Message Sequence Chart(MSC)*, Nov 1999. Series Z: Languages and general software aspects for telecommunication systems.
19. J. Klose and H. Wittke. An Automata Based Interpretation of Live Sequence Charts. In *TACAS*, pages 512–527, 2001.
20. P. Kosiuczenko and M. Wirsing. Formalizing and Executing Message Sequence Charts via Timed Rewriting. *Electr. Notes Theor. Comput. Sci.*, 25:1–25, 1999.
21. K. Koskimies and E. Mäkinen. Automatic Synthesis of State Machines from Trace Diagrams. *Softw. Pract. Exper.*, 24(7):643–658, 1994.
22. X. S. Li, Z. M. Liu, and J. F. He. A Formal Semantics of UML Sequence Diagram. In *Australian Software Engineering Conference*, pages 168–177. IEEE Computer Society, 2004.
23. R. Marelly and H. Kugler. Multiple Instances and Symbolic Variables in Executable Sequence Charts. In *Proceedings of OOPSLA'02*, pages 83–100, 2002.
24. A. Pnueli and R. Rosner. Distributed Reactive Systems are Hard to Synthesis. In *Proc. of 31st IEEE Sypm. on Foudation of Computer Science*, 1990.
25. A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice-Hall, 1997.
26. S. Schneider, J. Davies, D. M. Jackson, G. M. Reed, J. N. Reed, and A. W. Roscoe. Timed CSP: Theory and practice. In *Real-Time: Theory in Practice*, volume 600, pages 640–675. Springer-Verlag, 1992.
27. J. Sun and J. S. Dong. Model Checking Live Sequence Charts. In *ICECCS'05, to appear*, 2005.

Appendix A: Proof of L8

Given two processes $\widetilde{\mathcal{P}_1}$ and $\widetilde{\mathcal{P}_2}$ and their alphabets X, Y respectively, the alphabetized parallel composition is denoted as $\widetilde{\mathcal{P}_1} \ {}_X||_Y \ \widetilde{\mathcal{P}_2}$.

$$\widetilde{\mathcal{P}_1}\ {}_X||_Y\ \widetilde{\mathcal{P}_2} = \{(u, M \cup N) : (X \cup Y)^* \times \mathbb{P}(X \cup Y) \mid u \in traces(\widetilde{\mathcal{P}_1}\ {}_X||_Y\ \widetilde{\mathcal{P}_2}) \land M \setminus (X \cap Y) = N \setminus (X \cap Y) \land (u \upharpoonright X, M) \in \widetilde{\mathcal{P}_1} \land (u \upharpoonright Y, N) \in \widetilde{\mathcal{P}_2}\} \quad \text{[Def1]}$$

An event sequence u is a trace of $\widetilde{\mathcal{P}_1}\ {}_X||_Y\ \widetilde{\mathcal{P}_2}$ if and only if there exists a trace $s : traces(\widetilde{\mathcal{P}_1})$ and $t : traces(\widetilde{\mathcal{P}_2})$ such that $u \in s\ {}_X||_Y\ t$. The definition of $s\ {}_X||_Y\ t$ can be referred in [25].

Lemma 1. $\forall (s, E) \bullet (s, E) \in \mathcal{F}(\widetilde{\mathcal{P}_1})\ {}_X||_Y\ \mathcal{F}(\widetilde{\mathcal{P}_2}) \Rightarrow (s, E) \in \mathcal{F}(\widetilde{\mathcal{P}_1}\ {}_X||_Y\ \widetilde{\mathcal{P}_2})$

Proof.

$$\begin{aligned}
&(s, M \cup N) \in \mathcal{F}(\widetilde{\mathcal{P}_1})\ {}_X||_Y\ \mathcal{F}(\widetilde{\mathcal{P}_2}) \\
&\Rightarrow s \in trace(\mathcal{F}(\widetilde{\mathcal{P}_1})\ {}_X||_Y\ \mathcal{F}(\widetilde{\mathcal{P}_2})) \land M \setminus (X \cap Y) = N \setminus (X \cap Y) \\
&\qquad \land (s \upharpoonright X, M) \in \mathcal{F}(\widetilde{\mathcal{P}_1}) \land (s \upharpoonright Y, N) \in \mathcal{F}(\widetilde{\mathcal{P}_2}) && \text{[Def1]} \\
&\Rightarrow s \in trace(\widetilde{\mathcal{P}_1}\ {}_X||_Y\ \widetilde{\mathcal{P}_2}) \land M \setminus (X \cap Y) = N \setminus (X \cap Y) \\
&\qquad \land (s \upharpoonright X, M) \in \widetilde{\mathcal{P}_1} \land \exists (s \upharpoonright X, M') : \widetilde{\mathcal{P}_1} \bullet \widehat{\Sigma} \subseteq M' \\
&\qquad \land (s \upharpoonright Y, N) \in \widetilde{\mathcal{P}_2} \land \exists (s \upharpoonright Y, N') : \widetilde{\mathcal{P}_2} \bullet \widehat{\Sigma} \subseteq N' && \text{[Def. of } \mathcal{F}] \\
&\Rightarrow (s, M \cup N) \in \widetilde{\mathcal{P}_1}\ {}_X||_Y\ \widetilde{\mathcal{P}_2} \\
&\land \exists (s, M' \cup N') : \widetilde{\mathcal{P}_1}\ {}_X||_Y\ \widetilde{\mathcal{P}_2} \bullet \widehat{\Sigma} \subseteq M' \cup N' \\
&\Rightarrow (s, M \cup N) \in \mathcal{F}(\widetilde{\mathcal{P}_1}\ {}_X||_Y\ \widetilde{\mathcal{P}_2}) && \text{[Def. of } \mathcal{F}]
\end{aligned}$$

Intuitively, this lemma states that if both components cannot engage a signal (all signals are refused) at certain point of execution, then the composition cannot engage a signal either. Unfortunately, the reverse is not true. The following illustrates a counter example where $\widetilde{\mathcal{P}_1}$ can be extended by engaging a shared signal and $\widetilde{\mathcal{P}_2}$ cannot.

$$\begin{aligned}
&\mathcal{P}_1 \mathrel{\widehat=} (\widehat{a} \rightarrow \text{STOP} \ \Box\ b \rightarrow \text{STOP}) && X \mathrel{\widehat=} \{\widehat{a}, b\} \\
&\mathcal{P}_2 \mathrel{\widehat=} c \rightarrow \text{STOP} && Y \mathrel{\widehat=} \{\widehat{a}, c\}
\end{aligned}$$

It is easy to verify that $(\langle\rangle, \{\widehat{a}\})$ is in $\mathcal{F}(\widetilde{\mathcal{P}_1}\ {}_X||_Y\ \widetilde{\mathcal{P}_2})$ but not $\mathcal{F}(\widetilde{\mathcal{P}_1})\ {}_X||_Y\ \mathcal{F}(\widetilde{\mathcal{P}_2})$. However, if we assume that whenever the two components agree on the set of refused local events, they also agree on the set of shared events, i.e. if one component is ready to engage a shared event, the other is ready too and vice versa, then the reverse is true. Formally, we assume

$$\forall (s, M) \in \widetilde{\mathcal{P}_1} \land (t, N) \in \widetilde{\mathcal{P}_2} \bullet M \setminus (X \cap Y) = N \setminus (X \cap Y) \Rightarrow M = N$$

Lemma 2. $\forall (s, E) \bullet (s, E) \in \mathcal{F}(\widetilde{\mathcal{P}_1}\ {}_X||_Y\ \widetilde{\mathcal{P}_2}) \Rightarrow (s, E) \in \mathcal{F}(\widetilde{\mathcal{P}_1})\ {}_X||_Y\ \mathcal{F}(\widetilde{\mathcal{P}_2})$

Proof.

$$\begin{aligned}
&(s, M \cup N) \in \mathcal{F}(\widetilde{\mathcal{P}_1}\ {}_X||_Y\ \widetilde{\mathcal{P}_2}) \\
&\Rightarrow (s, M \cup N) \in (\widetilde{\mathcal{P}_1}\ {}_X||_Y\ \widetilde{\mathcal{P}_2}) \land \exists (s, E') : (\widetilde{\mathcal{P}_1}\ {}_X||_Y\ \widetilde{\mathcal{P}_2}) \bullet \widehat{\Sigma} \subseteq E' && \text{[Def. of } \mathcal{F}] \\
&\Rightarrow s \in trace(\widetilde{\mathcal{P}_1}\ {}_X||_Y\ \widetilde{\mathcal{P}_2}) \land M \setminus (X \cap Y) = N \setminus (X \cap Y) \\
&\qquad \land (s \upharpoonright X, M) \in \widetilde{\mathcal{P}_1} \land (s \upharpoonright Y, N) \in \widetilde{\mathcal{P}_2} \\
&\qquad \land \exists (s, E') : (\widetilde{\mathcal{P}_1}\ {}_X||_Y\ \widetilde{\mathcal{P}_2}) \bullet \widehat{\Sigma} \subseteq E' && \text{[Def1]}
\end{aligned}$$

From the healthiness conditions of CSP, there exists $(s \upharpoonright X, M')$ in $\widetilde{\mathcal{P}_1}$, and $(s \upharpoonright Y, N')$ in $\widetilde{\mathcal{P}_2}$ with maximal refusal set. By the definition of the alphabetized parallel composition, $M' \setminus (X \cap Y) = N' \setminus (X \cap Y)$ and, therefore, by our assumption $M' = N' = E'$.

$$\Rightarrow s \in trace(\widetilde{\mathcal{P}_1}\ {}_X||_Y\ \widetilde{\mathcal{P}_2}) \wedge M \setminus (X \cap Y) = N \setminus (X \cap Y)$$
$$\wedge\ (s \restriction X, M) \in \widetilde{\mathcal{P}_1} \wedge (s \restriction Y, N) \in \widetilde{\mathcal{P}_2}$$
$$\wedge\ (s \restriction X, E') \in \widetilde{\mathcal{P}_1} \wedge \widehat{\Sigma} \subseteq E' \wedge (s \restriction Y, E') \in \widetilde{\mathcal{P}_2} \wedge \widehat{\Sigma} \subseteq E' \quad \text{[By assump.]}$$
$$\Rightarrow s \in trace(\widetilde{\mathcal{P}_1}\ {}_X||_Y\ \widetilde{\mathcal{P}_2}) \wedge M \setminus (X \cap Y) = N \setminus (X \cap Y)$$
$$\wedge\ (s \restriction X, M) \in \mathcal{F}(\widetilde{\mathcal{P}_1}) \wedge (s \restriction Y, N) \in \mathcal{F}(\widetilde{\mathcal{P}_2}) \quad \text{[Def. of } \mathcal{F}\text{]}$$
$$\Rightarrow (s, E) \in \mathcal{F}(\widetilde{\mathcal{P}_1})\ {}_X||_Y\ \mathcal{F}(\widetilde{\mathcal{P}_2}) \quad \text{[Def1]}$$

Thus, by Lemma 1 and 2 we conclude **L8**. Law **L9** is a direct consequence of law **L8** and the symmetry and associativity laws of alphabetized parallel.