TEMPORAL SPECIFICATIONS OF CLIENT-SERVER SYSTEMS AND UNBOUNDED AGENTS

By

S. Sheerazuddin

The Institute of Mathematical Sciences, Chennai.

A thesis submitted to the
Board of Studies in Mathematical Sciences

In partial fulfillment of the requirements

For the Degree of

DOCTOR OF PHILOSOPHY

of

HOMI BHABHA NATIONAL INSTITUTE

January 2013

Homi Bhabha National Institute

Recommendations of the Viva Voce Board

As members of the Viva Voce Board, we recommend that the dissertation prepared by **S. Sheerazuddin** entitled "Temporal Specifications of Client-Server Systems and Unbounded Agents" may be accepted as fulfilling the dissertation requirement for the Degree of Doctor of Philosophy.

Date :
Convener : R. Ramanujam

Date :
Member : V. Arvind

Date :
Member : Kamal Lodaya

Date :
Member : S. P. Suresh

Date :
External Examiner : Paritosh K. Pandya

Final approval and acceptance of this dissertation is contingent upon the candidate's submission of the final copies of the dissertation to HBNI.

I hereby certify that I have read this dissertation prepared under my direction and recommend that it may be accepted as fulfilling the dissertation requirement.

Date :
Guide : R. Ramanujam

DECLARATION

I declare that the thesis titled TEMPORAL SPECIFICATIONS OF CLIENT-SERVER SYSTEMS AND UNBOUNDED AGENTS submitted by me for the degree of Doctor of Philosophy is the record of work carried out by me during the period from 2005 to 2011 under the guidance of **Prof. R Ramanujam**. The work is original and has not been submitted earlier as a whole or in part for a degree/diploma at this or any other Institution or University.

S. Sheerazuddin

to the women in my life. . . . . ,
kausar jahan, sanober yasmin and shazia samreen.

ACKNOWLEDGEMENTS

I would, first and foremost, like to thank my advisor Prof. R. Ramanujam for his guidance and support all these years. Jam, as we call Prof. Ramanujam, has been the prime mover behind this work, from its inception to the end. Without Jam this thesis could have never seen the light of day. Jam continued pushing me, with patience, with all the encouragement I needed and, now I have a thesis in hand. It seems like a dream come true and, undoubtedly, all the credit must go to Jam.

Jam has also been an all-weather friend and guide, in the mould of the sadguru of kabir's poetic universe, for me. He has been an assuring presence, a sort of beacon of hope, that things will turn out to be alright at the end. He taught me that whatever be the situation, never give up and continue doing your assigned work with unflinching stoicism. Thanks a lot Jam.

"Guru and God both are here, to whom should I first bow
All glory be unto the guru, path to God who did bestow."

Kamal Lodaya and S. P. Suresh have been on my doctoral committee for the duration of my PhD. They have not only given invaluable suggestions for the thesis, but also taught me basic courses on distributed systems, programming languages and automata theory. I would like to thank them for their support and teaching. In particular, I would like to thank Kamal who took particular interest in my work and went out of way to give me much needed encouragement, support and help.

The TCS group at IMSc and CMI welcomed me as one of their own and guided us through an excellent course work through which I learnt the subject. I am grateful to all of them for their excellent teaching – V. Arvind, Kamal Lodaya, Madhavan Mukund, Meena Mahajan, K. Narayan Kumar, R. Ramanujam, Venkatesh Raman, C. R. Subramanian and Rani Siromoney.

I would also like to thank Prof. R. Balasubramanian, Director IMSc, who allowed me to stay longer than the duration of my fellowship. Thus, I could complete my work without any distraction and difficulty.

I would like to thank all those who shared my room and cooked and cleaned for me – Ataur rahman, Shamim, Danish, Hamid and Shakir. Special thanks are due to Najma, wife of Hamid, who fed me authentic Bangla food for many months.

Also, I would like convey my immense thanks to my friends at IMSc who made my life bearable in last eight years – Ram, Neeraj, Sudhir, Baskar, Amal, Abhinav, Bruno, Kamil bhai, Dhriti and many others. Ram has been a constant companion for last ten years, beginning from our REC days. He shared my room for a couple of years too. Special thanks are due to him.

At the end, I would like to convey my special gratitude to my parents, Shamimuddin and Kausar Jahan, for all their love and support. Thanks a lot, Papa and Ammi. Also, I would like to thank my siblings, Razi Ahmad, Rafi Ahmad, Sanober Yasmin, Merajuddin, Sarfarazuddin and Shahnawazuddin for being there through thick and thin. It was great growing up with you. How can I forget to thank Shazia Samreen, a remarkable woman, extremely patient and undemanding. Thanks a lot for choosing to be my wife.

Abstract

The Client-server model of computing is a distributed application structure that partitions tasks or workloads between service providers, called servers, and service requesters, called clients. In this thesis, we study the formal specification and verification of such client server systems. For convenience, we consider them to be of two kinds: Single Client Multiple Server Systems (SCMS) and Single Server Multiple Client Systems (SSMC).

In SCMS systems, a single client interacts with a host of servers, directly or indirectly, to obtain some service. The number of servers is fixed *a priori*. The use of formal methods for SCMS is concentrated mainly on reasoning about communication among the various servers. In particular, the challenge is to come up with appropriate logical languages to describe good (valid) patterns and with procedures for checking that all behaviours (computations) of a given system conform to these good patterns.

In [67], Meenakshi and Ramanujam propose a local temporal logic (m-LTL) in which such specifications can be written. The systems they study, called Systems of Communicating Automata (SCA), are a variant of CFSMs [16]. The computations of SCAs are a variant of MSCs called Lamport Diagrams, a class of partial orders generated by MSCs [77]. The model checking problem is shown to be decidable, without putting any bound on the channel capacity.

We explore the suitability of m-LTL to specify properties of SCMS systems, and find that two changes are appropriate. m-LTL uses an immediate past modality ($\ominus$), whereas the (transitive) "some time in the past" ($\diamondminus$) modality is found to be more appropriate for services. We advocate the need for a concurrent present modality, called now, to talk about the properties true in the (possibly) present state of some other agent in the system. We call this modified logic w-LTL.

The new modality $\langle now \rangle_j$ refers to the present. Over total orders, the present is a point, whereas over partial orders, the present is an interval. $\langle now \rangle_j \beta$, asserted by i, says that β holds in some j-local state concurrent with i's local state. We present a detailed specification of a Travel Agency Web Service illustrating the use of the new modality and distinguish these specifications from those in m-LTL.

The main theorem for w-LTL that we present is the decidability of satisfiability and model checking.

In SSMC systems, an unbounded number of clients of different types need the service of a single server. The number of client types is fixed beforehand. Here, the formal problem is to model systems with unboundedly many agents, which, naturally, gives rise to infinite state systems. Restricting the expressiveness of specification mechanisms so that we can still get decidable model checking becomes the focus of our attention in this case.

In the literature [22], SSMC systems appear in two flavours: discrete and session-oriented. In the case of discrete services, the client sends a request for service to the server and waits for the response, which can either be acceptance or rejection. On the other hand, in the session-oriented paradigm, the client interacts with the server between the send-request and receive-response, in some non-trivial way. This interaction, in turn, may affect the outcome of client request. We can say, that, in the discrete case the clients are passive, whereas, in the other case, the clients are active.

For each paradigm, we present an automaton model, System of Passive Clients (SPS) for discrete services and System of Active Clients (SAS), for session-oriented services. Our models have the desired capability, they allow for unbounded number of clients. Consequently, these are infinite space systems. As a result, their reachability properties are typically undecidable to check. In the thesis we show that SAS are equivalent to (or have the same behaviour as) multi-counter automata, whereas SPS is a subclass of SAS. The class of SAS machines have the same closure properties as class of counter machines with no zero test. In particular, SAS turn out to be closed under union and intersection but not under complementation. Also, they have a decidable reachability algorithm as given by Mayr, Lambert and Kosaraju [55] [58] [65]. On the other hand, once we bound the number of clients, they reduce to finite state machines. We can exploit this property to model check specifications against such models.

There are several candidate temporal logics for message passing systems, but these work with *a priori* fixed number of agents, and for any message, the identity of the sender and the receiver are fixed at design time. We need to extend such logics with means for referring to agents in some more abstract manner (than by name).

A natural and direct approach to refer to unknown clients is to use logical variables: Rather than work with atomic propositions p, we use monadic predicates

$p(x)$ to refer to property p being true of client x. We can quantify over such x existentially and universally to specify policies relating clients. We are thus naturally led to the realm of Monadic First Order Temporal Logics ($MFOTL$)[35]. In fact, it is easily seen that $MFOTL$ is expressive enough to frame almost every requirement specification of client-server systems. Unfortunately, $MFOTL$ is undecidable [45], and we need to limit the expressiveness so that we have a decidable verification problem. Hodkinson *et. al.*, in [45], show that allowing two or more free variables in the scope of a temporal modality in $MFOTL$ leads to undecidability, it can encode the $\mathbb{N} \times \mathbb{N}$ recurring tiling problem [39], [40]. They restrict $MFOTL$ to its monodic fragment, where there are at most one free variable in the scope of any temporal modality, and obtain decidable algorithm for satisfiability.

The logical language to specify and verify SPS-like systems has two mutually exclusive dimensions. One, defined by an MFO fragment, talks about the plurality of clients asking for a variety of services. The other, defined by an LTL fragment, talks about the temporal variations of services being rendered. Furthermore, the MFO fragment has to be multi-sorted to cover the multiplicity of service types. Keeping these issues in mind, we frame a logical language, which we call $\mathcal{L}_{SPS}$. $\mathcal{L}_{SPS}$ is a combination of LTL and multi-sorted MFO. In the case of LTL, atomic formulas are propositional constants which have no further structure. In $\mathcal{L}_{SPS}$, there are two kind of atomic formulas, basic server properties from P_s, and MFO-sentences over client properties P_c. Consequently, these formulas are interpreted over sequences of MFO-structures juxtaposed with LTL-models.

We show the satisfiability and model checking of $\mathcal{L}_{SPS}$ to be decidable. The proof uses a formula automaton construction, and in this sense, offers some novelty for a temporal logic with some (limited) quantification.

When we consider temporal specifications for requirements of SAS, we need to strengthen the logical language, one which we call $\mathcal{L}_{SAS}$. Note, that, closing MFO sentences with temporal modalities, as in the case of $\mathcal{L}_{SPS}$ is not enough for this case. Since the clients are engaged with the server for a period of nontrivial interaction, we need to refer to temporal instances. On the other hand, clients are denoted by free variables, and allowing more than one free variable in the scope of temporal modalities leads to undecidable logics. So, we need to consider an $MFOTL$ fragment with suitable constraints on the specifications in a way that they are expressive enough and, additionally, has a decidable verification algorithm.

We propose a fragment of monadic monodic temporal logic ([45]) as the specification language for SAS. In $\mathcal{L}_{SAS}$, the valid specifications hail from the following set:

$$\psi \in \Psi ::= q \in P_s \mid \neg q \mid (\exists x : u)\alpha \mid \psi_1 \vee \psi_2 \mid \psi_1 \wedge \psi_2 \mid \Diamond\psi \mid \Box\psi$$

where $u \in \Gamma_0$, the set of client types, and α's are client formulas defined as follows:

$$\Delta_x ::= p \in P_c \mid \neg\alpha \mid \alpha_1 \vee \alpha_2 \mid \Diamond\alpha.$$

Note that P_c is the set of local client propositions and P_s is the set of local server propositions. Since it is a fragment of a decidable logic (monadic monodic temporal logic), its satisfiability problem is decidable. We present a formula automaton construction, using a multi-counter automaton, that leads to a non-elementary decision procedure for the satisfiability of $\mathcal{L}_{SAS}$.

Contents

List of Figures

1
Introduction

Client-server model of computing is a distributed application structure that partitions tasks or workloads between service providers, called servers, and service requesters, called clients. Often clients and servers communicate over a computer network on separate hardware, but both client and server may reside in the same system. A server machine is a host that is running one or more server programs which share its resources with clients. A client does not share any of its resources, but request a server's content or service function. Clients therefore initiate communication sessions with servers which await (listen for) incoming requests.

Functions such as email exchange, web access and database access, are built on the client-server model. For example, a web browser is a client program running on a user's computer that may access information stored on a web server on the Internet. Users accessing banking services from their computer use a web browser client to send a request to a web server at a bank. That program may in turn forward the request to its own database client program that sends a request to a database server at another bank computer to retrieve the account information. The balance is returned to the bank database client, which in turn serves it back to the web browser client displaying the results to the user.

In this thesis, we study the formal specification and verification of client-server systems. For convenience, we consider them to be of two kinds: Single Client Multiple Server Systems (SCMS) and Single Server Multiple Client Systems (SSMC). In SCMS systems, a single client interacts with a host of servers, directly or indirectly, to obtain some service. The number of servers is fixed *a priori* . The use of formal methods for SCMS is concentrated mainly on reasoning about communication among the various servers. In particular, the challenge is to come up with appropriate logical languages to describe good (valid) patterns and with procedures for checking that all behaviours (computations) of a given system conform to these good patterns.

We consider SCMS systems mainly from the Web. We work with two examples, Travel Agency Web service and Quote Request Web service. Travel Agency WS is centred around a travel agent service which manages the travel requirements of a client. The travel agent interacts with other services, airline which handles flight booking, train which handles rail reservation and hotel which handles accommodation, and prepares the best possible package for the customer. Quote Request WS consists of a single buyer which is the client, a number of suppliers and manufacturers, which are the servers. The buyer interacts with the multiple suppliers in order to purchase some commodity, say cars or obtain some service, health insurance for employees. The suppliers, in turn, interact with the multiple manufacturers and prepare quotes for the buyer.

In SSMC systems, an unbounded number of clients of different types need the service of a single server. The number of client types is fixed beforehand. Here, the formal problem is to model systems with unboundedly many agents, which naturally gives rise to infinite state systems. Restricting the expressiveness of specification mechanisms so that we can still get decidable model checking becomes the focus of our attention in this case.

We consider two examples of SSMC systems, one is a variation of Travel Agency WS and another is the Loan approval WS. In Loan Approval WS, there is a designated Web Server acting as loan officer which admits loan requests of various sizes. Depending on the number of loan requests and their sizes and according to an *a priori* fixed loan disbursal policy, the loan officer accepts or rejects the pending requests. The modified Travel Agency WS consists of a travel agent service and two types of clients, hotel, h, and airline, a. The clients of type h look after accommodation needs in hotel/s, whereas those of type a offer bookings on airlines, as mentioned earlier. There are unboundedly many agents of each type competing to cater to the needs of the travel agent. The travel agent, in turn, has to come up with holiday packages, suitable to the needs and pockets of it's targeted customer base. This, it does by interacting with the competing h and a clients.

1.1 Preliminaries

SCMS systems are essentially distributed or concurrent systems with *a priori* fixed number of agents (or processes). Concurrent systems are defined as systems in which programs are designed as collections of interacting computational processes that may be executed in parallel [14]. Concurrent programs can be executed sequentially on a single processor by interleaving the execution steps of each computational process, or executed in parallel by assigning each computational process to one of a set of processors that may be close or distributed across a network.

A concurrent system is formally described as a transition system $TS = (S, \rightarrow, I)$ over a set of actions Σ, where S is a (non-empty) set of states, $I \subseteq S$ is the set of initial states and $\rightarrow \subseteq S \times \Sigma \times S$, a set of permitted transitions [61]. An execution (or behaviour, run) of the TS can be viewed as an infinite sequence of states from S:

$$\rho = s_0 s_1 s_2 \cdots \text{ , where } s_0 \in I \text{ and } \forall i \geq 0 \; s_i \overset{a_{i+1}}{\rightarrow} s_{i+1} \text{ for some } a_{i+1} \in \Sigma.$$

A property or specification α of TS is a set of such sequences. A property α holds for a concurrent system TS if the set of sequences defined by TS is contained in the property α. In the literature, it is considered useful to distinguish two kinds of properties; safety properties and liveness properties, as described by Leslie Lamport in [59]. Safety properties were formalized in [1] and liveness properties in [4]. The following discussion on formal definitions of safety and liveness properties is from [4].

1.1.1 Safety Properties

Informally, a safety property stipulates that some "bad thing" does not happen during execution. Examples of safety properties include mutual exclusion, partial correctness and first-come-first-serve *etc.* One way to formalize safety is as follows: Let S be the set of program states, and S^ω be the set of infinite sequences of program states, whereas S^* is the set of finite sequences. An execution of a program can be modelled as a member of S^ω. We call elements of S^ω executions and those of S^*, partial executions. Also, given a property α, and an execution ρ, ρ satisfying

α is denoted by $\rho \models \alpha$. Finally, let ρ_i denote the partial execution consisting of the first i states in ρ.

For α to be a safety property, if α does not hold then some "bad thing" must happen. Such a bad thing must be irremediable because a safety property says that the bad thing never happens during execution. Thus, α is a safety property if and only if

$$(\forall \rho)\rho \in S^{\omega} : \rho \not\models \alpha : \big((\exists i) : i \geq 0 : \big((\forall \rho')\rho' \in S^{\omega} : (\rho_i \cdot \rho') \not\models \alpha\big)\big).$$

There are two things to note about this definition. First, when a "bad thing" occurs during an execution then there is an identifiable point at which it happens. Second, the definition unconditionally prohibits a "bad thing" from occurring.

1.1.2 Liveness Properties

Informally, a liveness property stipulates that a "good thing" happens during execution. Examples of liveness properties include starvation freedom, termination and guaranteed service.

The thing to observe about a liveness property is that no partial execution is irremediable: it is always possible for the required "good thing" to occur in future. Liveness is formalized as follows: a partial execution ρ is live for a property α if and only if there is a sequence of states ρ' such that $\rho \cdot \rho' \models \alpha$. A liveness property is one for which every partial execution is live. Thus α is a liveness property if and only if

$$(\forall \rho)\rho \in S^{*} : \big((\exists \rho')\rho' \in S^{\omega} : (\rho \cdot \rho') \models \alpha\big).$$

Again, there are two things to note about the definition. First, the definition does not restrict what a "good thing" can be. In this way, liveness is fundamentally different from safety. Second, a liveness property cannot stipulate that some "good thing" always happens, only that it eventually can happen.

1.1.3 Other Properties

Many properties are neither safety nor liveness. For example, any property characterized by **until**: "Eventually an event of type e_2 will happen and all preceding events are of type e_1", is not, *prima facie*, of either type.

Plotkin is credited to have shown [4], using a topological argument, that every property is the intersection of a safety property and a liveness property. For example, the until property, given above, is the intersection of safety property "$\neg e_1$ before e_2 does not happen" and the liveness property "e_2 eventually happens". Total correctness is also the intersection of a safety property (partial correctness) and liveness property (termination).

1.1.4 Specification Language

Temporal logics are the standard logical formalisms to express safety and liveness properties. They were first proposed to be used for concurrent systems by Amir Pnueli in his seminal paper [72]. Temporal logics come in two variants, linear time and branching time. Linear time logics are concerned with properties of paths of labelled transition systems (LTS). An LTS is transition system with states having labels, $V : S \to 2^P$, where P is a set of atomic propositions. A state in an LTS is said to satisfy a linear-time property if all paths emanating from this state satisfy the property. In an LTS, for example, two states that generate the same language satisfy the same linear-time properties. Branching-time logics, on the other hand, describe properties that depend on the branching structure of the LTS. Two states may generate the same language, but may often have different branching structures distinguishable by a branching-time formula.

Linear Time Temporal Logic

Propositional linear-time temporal logic (LTL) is the basic linear time logic. It is often represented in a form to be interpreted over labelled transition systems (LTS). Its formulae are constructed as follows, where p ranges over a set P of atomic propositions as already mentioned:

$$\Phi ::= p \mid \neg\alpha \mid \alpha \vee \beta \mid \bigcirc\alpha \mid \alpha\mathbf{U}\beta.$$

LTL formulae are interpreted over paths in an $LTS = (S, \to, I, V)$. A path is non-empty sequence $\rho = s_0 s_1 s_2 \cdots$ of states $s_0, s_1, s_2 \in S$ such that for all $i \geq 0$, $(s_i, s_{i+1}) \in\to$. For all $0 \leq i \leq |\rho|$, let $\rho[i]$ denote the i-th state in the path and ρ^i

the tail of the path starting at state $\rho[i]$, seen as $\rho[i] \cdot \rho[i+1] \cdots$. In particular, $\rho^0 = \rho$.

As the semantics of LTL, we define an inductive definition of when a path ρ in a LTS *satisfies* a formula α.

1. $\rho \models p$ iff $p \in V(\rho[0])$.
2. $\rho \models \neg\alpha$ iff $\rho \not\models \alpha$.
3. $\rho \models \alpha \vee \beta$ iff $\rho \models \alpha$ or $\rho \models \beta$.
4. $\rho \models \bigcirc\alpha$ iff $\rho^1 \models \alpha$.
5. $\rho \models \alpha\mathbf{U}\beta$ iff $\exists k \geq 0 : \rho^k \models \beta : \forall l : 0 \leq l < k : \rho^l \models \alpha$.

We can extend the definition of labels V to paths as follows and define the set of all models of an LTL formula α. Given a path $\rho = s_0 s_1 s_2 \cdots$ in labelled transition system $LTS = (S, \rightarrow, I, V)$, the label of the path ρ is: $V(\rho) = V(s_0)V(s_1)V(s_2)\cdots$. Now, for any LTL formula α we can define set of all models of α as the set $Models(\alpha) = \{V(\rho) \mid \rho \text{ is a path in some } LTS = (S, \rightarrow, I, V) \text{ and } \rho \models \alpha\}$.

Temporal logics describe the ordering of events in time without introducing time explicitly. They were developed by philosophers and linguists for investigating how time is used in natural language arguments. Most temporal logics have an operator like $\Box\alpha$ that is true in the present if α is always true in the future. The dual of $\Box$ is $\Diamond$. A formula $\Diamond\alpha$ is true only if α is true in the future. Clearly, the following equivalence holds:

$$\Diamond\alpha \equiv T\mathbf{U}\alpha,$$

$$\Box\alpha \equiv \neg\Diamond\neg\alpha.$$

It is trivial to see that a safety property can be expressed by a formula of the type $\Box\alpha$ and liveness property by a formula of type $\Diamond\alpha$.

Bisimulation Invariance

An interesting property which is satisfied by Temporal logics is **Bisimulation Invariance**. Bisimulation is a rich concept which appears in various areas of theoretical computer science. Its origin lies in concurrency theory, for instance see Milner [68], and in modal logic, see for example van Benthem [83].

Bisimulations were introduced by Park [71] as a small refinement of the behavioural equivalence defined by Hennessey and Milner in [41] between basic CCS processes (whose behaviour is a transition system).

Definition 1.1.1. *A binary relation* $\mathbf{R}$ *between states of a labelled transition system* $LTS = (S, \rightarrow, I, V)$*, over* $\Sigma = 2^P$*, is a* bisimulation *in case whenever* $(s, t) \in \mathbf{R}$*,*

1. *for all* $p \in P$*,* $p \in V(s)$ *iff* $p \in V(t)$*,*
2. *for all* $s' \in S$*, if* $s \rightarrow s'$ *then there is* $t' \in S$ *such that* $(s', t') \in \mathbf{R}$ *and* $t \rightarrow t'$ *and*
3. *for all* $t' \in S$*, if* $t \rightarrow t'$ *then there is* $s' \in S$ *such that* $(s', t') \in \mathbf{R}$ *and* $s \rightarrow s'$*.*

Simple examples of bisimulations are identity relation and the empty relation. Two states of a transition system s and t are **bisimilar**, written $s \sim t$, if there is a bisimulation $\mathbf{R}$ with $(s, t) \in \mathbf{R}$.

We can define bisimulation over states of different transition systems.

Definition 1.1.2. *Given two labelled transition systems* $LTS_1 = (S_1, \rightarrow_1, I_1, V_1)$ *and* $LTS_2 = (S_2, \rightarrow_2, I_2, V_1)$ *over* $\Sigma = 2^P$*, a non-empty relation* $\mathbf{R} \subseteq S_1 \times S_2$ *is a* bismulation *between* LTS_1 *and* LTS_2 *iff for all* $s_1 \in S_1, s_2 \in S_2$*,* $(s_1, s_2) \in \mathbf{R}$ *implies the following conditions:*

1. *for all* $p \in P$*,* $p \in V_1(s_1)$ *iff* $p \in V_2(s_2)$*,*
2. *if there exists* $t_1 \in S_1$ *such that* $s_1 \rightarrow_1 t_1$ *then there is* $t_2 \in S_2$ *such that* $(t_1, t_2) \in \mathbf{R}$ *and* $s_2 \rightarrow_2 t_2$ *and*
3. *if there exists* $t_2 \in S_2$ *such that* $s_2 \rightarrow_2 t_2$ *then there is* $t_1 \in S_1$ *such that* $(t_1, t_2) \in \mathbf{R}$ *and* $s_1 \rightarrow_1 t_1$*.*

If there is a bisimulation relation $\mathbf{R}$ between LTS_1 and LTS_2 such that $(s_1, s_2) \in \mathbf{R}$, then we write $(LTS_1, s_1) \sim (LTS_2, s_2)$ and say that LTS_1 and LTS_2 bisimulate each other. Another useful notion for relating two transition systems is bisimulation equivalence.

Definition 1.1.3. *Two transition systems LTS_1 and LTS_2 are* bisimulation equivalent *if there exists a bisimulation relation $\mathbf{R}$ between LTS_1 and LTS_2 such that the following conditions hold:*

1. *for every $s_1 \in I_1$ there exists $s_2 \in I_2$ such that $s_1 \mathbf{R} s_2$ and*

2. *for every $s_2 \in I_2$ there exists $s_1 \in I_1$ such that $s_1 \mathbf{R} s_2$.*

Informally, LTS_1 and LTS_2 are bisimulation equivalent if they can *bisimulate each other at their initial states.*

Let ρ_1 be a partial execution in LTS_1 and ρ_2 be a partial execution in LTS_2. ρ_1 is bisimulation equivalent to ρ_2, denoted $\rho_1 \sim \rho_2$, if $\forall i \geq 0$, $\rho_1[i] \sim \rho_2[i]$. Now, the following claim is easy to verify:

Claim 1.1.4. *For every $s_1 \in S_1, s_2 \in S_2$ if $s_1 \sim s_2$ then for every partial execution ρ_1 in LTS_1 with $\rho_1[0] = s_1$ there is a partial execution ρ_2 in LTS_2 with $\rho_2[0] = s_2$ such that $\rho_1 \sim \rho_2$ and vice versa.*

Given a pair of transition systems we are interested in whether or not two systems are equivalent under one or more properties expressible in logic LTL. We phrase this notion of equivalence as bisimulation invariance.

Definition 1.1.5. *An LTL property α of labelled transition systems is* bisimulation invariant *if the following holds:*

for every ρ_1 in LTS_1 and ρ_2 in LTS_2, if $\rho_1 \sim \rho_2$ then $\rho_1 \models \alpha$ iff $\rho_2 \models \alpha$.

Now, it is easy to see the following:

Theorem 1.1.6. *Every LTL formula $\alpha \in \Phi$ is bisimulation invariant.*

1.1.5 Formal Verification of Distributed Systems

Formal verification is the technique of proving in a formal, mathematical way that a program satisfies its requirement. The program and its requirement or specification are modelled using a mathematical language. Given a distributed system description M and a specification α we need to *formally* verify that M satisfies α. Traditionally there have been two basic techniques of *formal verification*: theorem proving and model checking.

In interactive theorem proving, both M and α are modelled as a set of formulae in some logical language. M satisfies α, denoted by $M \models \alpha$ if and only if α is the logical consequence of M.

In the last two decades model checking has emerged as a promising and powerful approach to automatic verification of systems. Roughly speaking, a model checker is a procedure that decides whether a given structure M is a model of a logical formula α *i.e.*, whether M satisfies α, abbreviated as $M \models \alpha$. Intuitively, M is an abstract model of the system in question, typically an LTS and α, stated in a temporal or modal logic, describes a desirable property. The model checker then provides a "push button" approach for proving that the system modelled by M enjoys this property. This automation together with the fact that *efficient* model checkers can be constructed for *powerful* logics, forms the attractiveness of model checking.

The Model Checking Problem is easy to state:

Definition 1.1.7. *Let M be an LTS. Let α be a formula of temporal logic (i.e., the specification). Find all states s of M such that for every execution ρ of M if $\rho[0] = s$ then $\rho \models \alpha$.*

We use the term model checking because we want to determine if the temporal formula α is true in the LTS M, *i.e.*, whether the structure M is a model for the formula α.

Emerson and Clarke [23] gave a polynomial time algorithm to solve the model checking problem for the logic CTL. Quille and Sifakis [74] independently solved the Model Checking problem at the same time. Vardi and Wolper [84] first proposed the use of ω-automata (automata over infinite words) for automatic verification.

Automata Based Model Checking

One of the major approaches to automated verification is the automata-theoretic approach, which underlies model checkers that can handle linear time specifications. The key idea underlying the automata theoretic approach is that, given an LTL formula α, it is possible to construct a finite state automaton A_α on infinite words that accepts precisely all computations that satisfy α.

A Büchi Automaton is a tuple $A = (\Sigma, S, I, \delta, G)$ where Σ is a finite alphabet, S is a finite set of states, $I \subseteq S$ is a set of initial states, $\delta \subseteq S \times \Sigma \times S$ is the transition relation and $G \subseteq S$ is a set of accepting (good) states. A run of A over an infinite word $w = a_1 a_2 \cdots$ is a sequence $\rho = s_0 s_1 s_2 \cdots$ where $s_0 \in I$ and $s_i \in \delta(s_{i-1}, a_i)$ for all $i \geq 1$. A run ρ is accepting if there is some accepting state that repeats infinitely often. An infinite word w is accepted by A if there is an accepting run of A over w. The language of infinite words accepted by A is denoted by $L(A)$. The following theorem establishes the correspondence between LTL and Büchi automata.

Theorem 1.1.8. *Given an LTL formula α, let P_α be the set of all atomic propositions occurring in α. Then, one can build a Büchi Automaton $A_\alpha = (\Sigma, Q, I, \delta, G)$ where $\Sigma = 2^{P_\alpha}$ and $|Q| \leq 2^{O(|\alpha|)}$, such that $L(A_\alpha) = Models(\alpha)$.*

This correspondence reduces the verification problem to an automata-theoretic problem as follows: Suppose that we are given an abstraction of the system to be checked as a Büchi Automaton M and an LTL formula α. We check whether $L(M) \subseteq Models(\alpha)$ as follows:

1. construct the automaton $A_{\neg\alpha}$ that corresponds to the negation of the formula α; This automaton is called complementary automaton,
2. take the cross product of the system M and $A_{\neg\alpha}$ to obtain an automaton $L(A_{M,\alpha}) = L(M) \cap L(A_{\neg\alpha})$, and
3. check whether the language $L(A_{M,\alpha})$ is empty.

Theorem 1.1.9. *Let M be a Büchi Automaton and α be an LTL formula then M satisfies α, i.e., $M \models \alpha$ iff $L(A_{M,\alpha}) = \emptyset$.*

If $L(A_{M,\alpha})$ is empty then the design is correct. Otherwise, the design is incorrect and the word accepted by $L(A_{M,\alpha})$ is a computation of M which violates the property α. Such a computation is called counter-example in model checking parlance. A model checking tool analyzes the counter-example for feasibility *i.e.*, the violation is genuine or the result of an incomplete abstraction via M. If the violation is feasible, it is reported to the user; if it is not, the counter-example is used to refine M and the model checking is applied again.

Once the automaton $A_{\neg\alpha}$ is constructed, the verification task is reduced to automata-theoretic problems, namely, intersecting automata and testing emptiness of language of automata, which have efficient solutions. Further, using data structures that enable compact representation of very large state spaces makes it possible to verify designs of significant complexity.

1.2 Specification and Verification of Single Client Multiple Server Systems

SCMSs, in particular those studied in this thesis, Travel Agency and Quote Request Web service systems, are more naturally captured by a set of transition systems communicating asynchronously by exchanging messages across channels (FIFO queues). Let us call them message passing transition systems ($MPTS$). The channels may have some fixed capacity or they may be unbounded. In the thesis we use channels and queues interchangeably. The individual transition systems (agents) of the $MPTS$ typically do not share common actions and proceed with their computations in an autonomous fashion. The sender can put its message into the queue meant for the receiver (if it is not full) and proceed with its computation without having to wait for the receiver. However, computations can get blocked while waiting for a particular message or an agent might get stuck while trying to send a message through a queue whose capacity is already full.

An $MPTS$ with n agents can be described as a set of n peers where each peer is modelled as a Büchi automaton along with an input queue for incoming messages for every other peers. A peer i can send message to peer j by putting that message in the queue earmarked for j from where it can be read by j. Also, fix $\mathbb{M}$ as a finite set of messages. The queues are not explicit in the definition but, they play a crucial role in the computations of MPTS. Here, we use peer and agent interchangeably.

Definition 1.2.1. *An $MPTS$ is an n-tuple $A = (A_i, \cdots, A_n)$ where for each i, $1 \leq i \leq n$, the ith peer is a nondeterministic Büchi automaton $A_i = (\Sigma_i, S_i, I_i, \delta_i, G_i)$. The peer alphabet $\Sigma_i = \Gamma_i \cup (\mathbb{M} \times \{i\} \times \{!, ?\} \times (\{1, 2, \cdots, n\} - \{i\})$ where Γ_i is the set of i-local actions.*

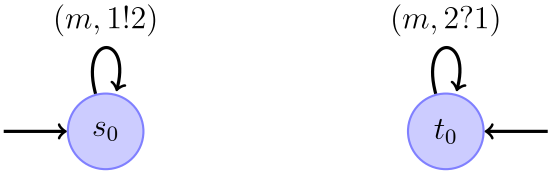


Figure 1.1: A simple $MPTS$

In a single transition, the agent i either makes a local move labelled by an element from Γ_i, or receives a message m from some agent j, labelled by $(m, i?j)$ or sends a message m to some agent j, labelled by $(m, i!j)$.

As an example, consider the $MPTS$ corresponding to the producer-consumer protocol. This $MPTS$ consists of two agents, 1 which is the producer and 2 which is the consumer. The producer generates a message m and puts it in the input queue of the consumer. The consumer removes the corresponding message from its queue. Here, $\mathbb{M} = \{m\}$ and Γ_1 as well as Γ_2 are empty. The computations of $MPTS$ can be modelled as partial order based diagrams, for example, the computation of producer-consumer $MPTS$ is given in the Figure 1.2. In this diagram, $e_1, e_2, e_3, \cdots$ are the events of producer, each of which are labelled by $(m, 1!2)$ and $f_1, f_2, f_3, \cdots$ are events of consumer which are labelled by $(m, 2?1)$.

In general, an n-agent diagram is a tuple $Dg = (E_1, E_2, \cdots, E_n, \lambda)$ where $E_1, E_2, \cdots, E_n$ are sets of events of the agents. Let $E = \bigcup\limits_i E_i$ be the set of all events then $\lambda : E \rightarrow E$ is a 1-1 function mapping sends to the receives. For each i, $1 \leq i \leq n$, there is an implicit total order over E_i.

The run ρ of an $MPTS$ A over a diagram Dg labels the (finite) configurations of Dg by a global state of A which contains two things, a tuple from $A_1 \times A_2 \times \cdots \times A_n$, which gives the local state of each agent, and the contents of each queue in that global state. A diagram Dg is accepted if each agent i accepts locally, which means visits its local good states G_i infinitely often and emptying its input queue, when E_i is infinite, otherwise it terminates in a good state with empty input queues. The language of an $MPTS$ is the set of all diagrams accepted by it. We shall describe these ideas, in detail, in the following chapters in the context of Sequence of Communicating Automata (SCA), an automaton model for SCMS systems, and Lamport diagrams which model computations of SCAs. Lamport diagrams are similar to diagrams described above except that λ is replaced by a relation which is designed to capture multiple sends and receives.

The properties (safety, liveness *etc.*) of $MPTS$ can now be described in terms of sets of diagrams. On the other hand, looking at the global product automaton of an $MPTS$ A, we find that it is of infinite size if the channel capacities are not *a priori* bounded. The details can be found in chapter 3. Clearly, checking language emptiness for such systems is, in general, undecidable. We can apply formal methods for these systems in one of the two ways, bound the channels, which reduces the $MPTS$ to a simple TS and then use the standard techniques with LTL, or use local logics.

1.2.1 Local Logics as Specification Languages

In order to specify $MPTS$, the traditional option is to consider the set of all sequentializations of all the diagrams representing the behaviour of a system and use standard LTL to reason about their properties. Sequentializations of a diagram are obtained by topologically sorting the events which respects the partial order (more about sequentializations in the next chapter). There are many drawbacks in such an approach. Firstly, even simple diagrams have sequentializations which do not form a regular language. For example, consider the diagram corresponding to the producer-consumer protocol given in Figure 1.2. The producer repeatedly sends messages labelled $(m, 1!2)$ to the consumer who receives them as $(m, 2?1)$. The set of all finite sequentializations of this diagram yields the language L over $(m, 1!2), (m, 2?1)$ where every word w in L is such that every prefix of it has at least as many $(m, 1!2)$'s as $(m, 2?1)$'s, which is not a regular language. Now, LTL cannot express such non-regular behaviours and so the diagrams specified would have to exclude such behaviours. The second drawback is that a logic like LTL specifies how the global states of the system may evolve whereas it would be ideal to have a logic which specifies the effect of message passing in the system. This is more naturally done using an event based approach instead of considering sequentializations. Such an event based logic would be able to specify properties by using the partially ordered structure of a diagram and formulae of logic can then talk about properties like when a particular agent can send a message, what would an agent do while receiving a message *etc.*

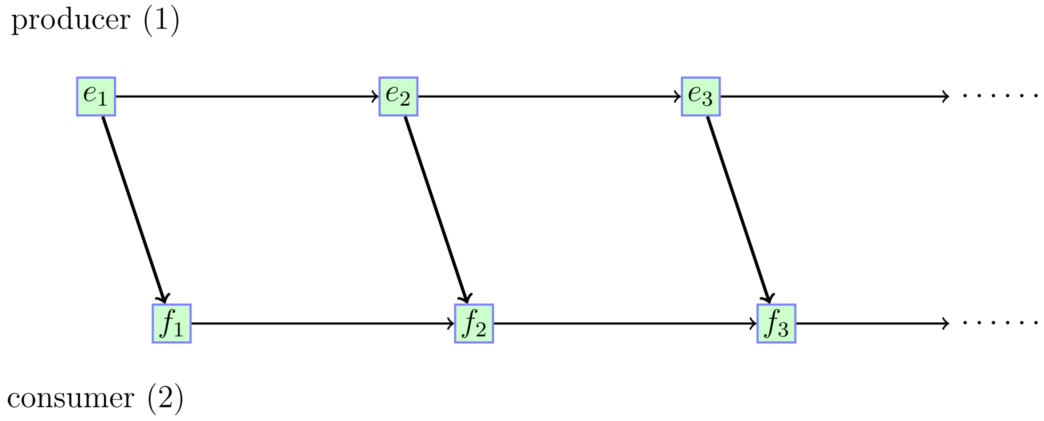


Figure 1.2: Diagram of the producer-consumer protocol

According to [63] a local logic for concurrent systems has the following important features:

1. The formulae and the structures for the logic reflect the fact that a system is composed out of a number of participating sequential agents,
2. formulae of the logic are interpreted only at local states, and
3. An agent makes a definite assertion about another agent only if it has received–directly or indirectly–some communication from that agent supporting that assertion.

One of the earliest instances of local logics was proposed in 1986 by Reif & Sistla [76]. The multiprocessor network logic had local temporal modalities and global spatial modalities named somewhere, everywhere and link. The formulae of this logic were interpreted over networks of arbitrary number of processes joined together through an incomplete set of links. The behaviour of individual processes was modelled by infinite sequences over a finite set of states. A formula was asserted at the ith local state in the behaviour of a particular process. They showed that multiprocess network logic had undecidable satisfiability as well as finite satisfiability. However, the model checking problem with finite network sizes was shown to be $PSPACE$-complete.

In [63] the authors propose a local linear time temporal logic with future and past modalities for Asynchronous Communicating Sequential Agents (ACSAs), a class of distributed systems. The semantics of the logic is chosen so that the modalities represent the notion of an agent gaining information about another through the reception of messages but not by sending them. In the same paper [63], they also showed completeness results of this logic over ACSAs and many subclasses of ACSAs. In [62], these (completeness) results were further improved upon. In [64], the logic defined in [62] is proved to be elementary decidable. This leads to decidability results for the logics over subclasses of ACSAs studied in [63].

A model checking algorithm for a variant of the logic introduced in [63] was described in [49]. The authors consider a variant of the logic in [63] and show that the problem of checking whether an asynchronous distributed net system whose behaviour is described by ACSAs, satisfies a given formula is decidable. However, the complexity of the algorithm is non-elementary in the size of the formula and the satisfiability problem too remains unaddressed.

In [75] the author proposes local linear time temporal logics of multiple agents with decidable satisfiability and model checking properties, albeit in a synchronous setting. Temporal assertions of agents refer to their local time and global assertions put them together. The agents refer to past, present and future of other agents, depending on their current view of the system, which in turn changes with communication. The logic has, apart from the usual temporal modalities, two novel modalities asserted in agent i, $\ominus_j$ referring to the most recent j-state in the past of agent i, and $\langle now \rangle_j$ referring to any j state in the unknown present of agent i. Interestingly, present tense can be a modality only in a partial order based logic as the ones in [75]. The frames of the logics are systems consisting of a fixed number of sequential components that synchronize by performing common actions together. Each component is a finite state automaton, and when a common (handshake) action is performed, it is a lock-step transition involving several automata at once.

In [67] the logic with $\ominus_j$ modality, defined in [75], interpreted over models with asynchronous communication is shown to be decidable using the standard automata theoretic techniques [84]. These models are referred to as Lamport Diagrams and are closely related to the standard formalism of Message Sequence Charts given by ITU norm Z. 120. The authors, in [67], solve the model checking problem of the logic, called m-LTL, against Sequence of Communicating Automata, a class of automata inspired by Communicating Finite State Machines [16].

1.3 Specification and Verification of Multiple Client Single Server Systems

SSMC systems, where one server and an unbounded number of clients collaborate, are akin to an infinite (parameterized) family of finite state systems. Such systems are parameterized because the number is known only at run time. Such a family can usually be seen as as one single infinite-state system. Checking reachability in parameterized systems is, in general, undecidable [8].

The verification problem for a family of similar state-transition systems is easy to formulate:

Given a family $M = \{M_i\}_{i=1}^{\infty}$ of systems M_i and a temporal formula α, verify that for every i, M_i satisfies α. This is known as the Parameterized Model-checking Problem (PMCP).

One way to define a family of parameterized synchronous systems is given in [32] as follows: the system instances are formed by control process C and copies of a generic user process U represented as (C, U) *family*. C and U are specified as finite-state labelled transition graphs.

The system instance of size n, M_n, is a synchronous parallel composition of C with n copies of process U, and is denoted as $C \parallel U^n = C \parallel U_1 \parallel U_2 \parallel \cdots \parallel U_n$. For all $1 \leq i \leq n$, U_i is a copy of U obtained by uniformly subscripting the states of U with i. Thus, for all $1 \leq i, j \leq n$, U_i and U_j are isomorphic upto re-indexing.

Transitions in C and U are labelled with guards. Every guard is a boolean combination of *user conditions* which have the form $(\exists i)\mathcal{E}(i)$. $\mathcal{E}(i)$ is a boolean expression formed from atomic propositions over the states of U_i and over states of C.

$\mathcal{G}_n$ denotes the global state transition system of the instance of size of n. A state s of $\mathcal{G}_n$ is written as an $(n+1)$-tuple $(c, u_1, \cdots, u_n)$ where c is a local state of C and the $(i+1)$th component of the tuple is a local state of U_i (for $i \in \{1, \cdots, n\}$). The initial state of $\mathcal{G}_n$ is $(i_C, (i_U)_1, \cdots, (i_U)_n)$. Given a global state $s = (c, u_1, \cdots, u_n)$, the local states are obtained as follows: $s(0) = c$ and $\forall i, 1 \leq i \leq n$, $s(i) = u_i$. A transition (s, t) is in $\mathcal{G}_n$ iff

1. A transition of C from $s(0)$ to $t(0)$ is enabled in s, and

2. For all $i \in \{1, 2, \cdots, n\}$, a transition of U_i from $s(i)$ to $t(i)$ is enabled in s,

where a transition of a process is said to be enabled in a global state iff the corresponding guard is true when evaluated in that global state. For a global state s and a guard g, $s \models g$ is defined inductively with base case as follows: $s \models (\exists i)\mathcal{E}(i)$ iff for some $k \in \{1, \cdots, n\}$ $\mathcal{E}(k)$ is true given the propositions that hold at $s(0)$ (the control state) and $s(k)$ (state of the user process U_k).

As mentioned above, the above problem (PMCP) is undecidable, in general. However, for *specific* families the problem may be solvable.

Browne et al. [17] consider the problem of verifying a family of token rings, that is, the family of rings of size 2, size 3, size 4 and so on. In order to verify the entire family, they establish a *bisimulation* relation between a two-process ring and an n-process token ring for any $n \geq 2$. It follows that the two-process ring and the n-process ring satisfy exactly the same temporal formulae. The drawback of their technique is that the bisimulation relation has to be constructed manually.

Emerson & Namjoshi [31] and German & Sistla [80], have shown that it is possible to automatically solve the parameterized model checking problems for some special cases. They prove that for rings composed of certain kind of processes there exists a k such that the correctness of the ring with k processes implies the correctness of rings of arbitrary size.

In [30], the authors consider the Parameterized Model Checking Problem (PMCP) for systems consisting of processes arranged in a ring that communicate by passing messages via tokens whose values can be updated *at most a bounded number of times*. Correctness properties are expressed using the stuttering-insensitive linear time logic $LTL \setminus \bigcirc$. For bidirectional rings they show how to reduce reasoning about rings with an arbitrary number of processes to rings with up to a certain finite cutoff number of processes. This immediately yields decidability of the PMCP. They go on to show that for unidirectional rings smaller cutoffs can be achieved, making the decision procedure provably efficient.

Looking at the big picture, [25], [79] and [87] propose a technique for uniform verification of parameterized systems called method of network invariants. The family of state-transition systems is represented by a context-free network grammar. Using the structure of the network grammar the given technique constructs an *invariant* $\mathcal{I}$ which *simulates* all the state-transition systems in the family. The parameterized system $M = \{M_i\}_{i \geq 1}$ satisfies a property α if $M \parallel \mathcal{I}$ satisfies α. Recently, in [53], the authors have improved the results of [87] and [57] by taking into account the fairness properties of the compared systems. Consequently, the abstraction can support and simplify proofs of liveness properties as well as any other property expressible by LTL.

In [73], Pnueli *et al.* introduce the $(0, 1, \infty)$-counter abstraction method by which a parameterized system of unbounded size is abstracted into a finite-state system. As each process in the parameterized system is finite-state, the abstract variables are limited counters which count, for each local state s of a process, $\kappa(s)$–the number of processes which currently are in local state s. The counters are saturated at 2, which means that $\kappa(s) = 2$ whenever 2 *or more* processes are at state s. The emphasis is on the derivation of an *adequate and sound* set of fairness requirements (both weak and strong) that enable proofs of liveness properties of the abstract system, from which we can safely conclude a corresponding liveness property of the original parameterized system.

In [89], Pnueli & Zuck explore two families of methods for automated verification of parameterized systems. One family, called the *method of auxiliary constructs*, automates the user-supplied interactive steps in deductive verification (the method of theorem proving). The other, *counter abstraction*, abstracts the parameterized system and its specifications, so that its verification is reduced to simple model checking. The premise underlying both the approaches is that the "infiniteness" of such systems is restricted to the number of processes, since each process is usually finite-state. The method of auxiliary constructs exploits the inherent symmetry of a system to derive inductive assertions about it. Counter abstraction uses the relative small size of the individual modules to construct finite-state abstraction.

In [82], the authors consider the verification of parameterized boolean programs. They propose that such programs can be model-checked by iteratively considering the program under k-round schedules, for increasing value of k, using a compositional construct called *linear interfaces* that summarize the effect of a block of threads in a k-round schedule. They also develop a game-theoretic *sound* technique to show that k rounds of schedule suffice to explore the entire search-space, which allows to prove a parameterized program entirely correct.

Similarly, in [9] the authors propose a theoretical solution to the model checking problem of reachability in concurrent programs with dynamic creation of threads, where a thread is context-switched into only a bounded number of times. They show reductions between this reachability problem and Petri-net coverability.

SSMC are also similar to those procedures which lead to potentially unbounded call stacks, namely pushdown systems. In such systems, process creation gives rise to unboundedly many processes and, therefore, infinitely many states. In [15], (possibly infinite) sets of configurations of pushdown systems are represented by means of finite-state automata and the authors give a procedure to compute sets of predecessors (of configurations) using this representational structure. This machinery is used to define model checking algorithms for pushdown systems against both linear time and branching properties.

1.3.1 Specification Languages

The most natural specification languages for parameterized systems would be parameterized temporal logics. In [17], the authors use Indexed $CTL^*(ICTL^*)$ which includes all of CTL^* [24] except the $\bigcirc$ operator and, additionally, formulae of the forms $\bigvee_i f(i)$ and $\bigwedge_i f(i)$. The subformula $f(i)$ is a so called *generic* formula; all the atomic propositions that appear within it must be subscripted by i.

In [36], the authors use Indexed LTL, similar to $ICTL$, to describe the properties of systems with a single control process C and an arbitrary number of user processes U. They also show that model checking is undecidable for $ILTL$ with respect to (C, U).

In order to specify a network of LTSs with a finite S composed of arbitrary number of components, [26] uses universal branching time temporal logic [27] with *regular languages* over finite states S as atomic formulae. Note that an arbitrary global state $\widetilde{s}$ in the network of LTS is a tuple from S^i, for some $i \geq 0$, where $S^i = S \times S \times \;\; i$ times. We can view $\widetilde{s}$ as a word in S^*. Let $\mathcal{D}$ be a finite state automaton defined over S. We say, global state $\widetilde{s}$ satisfies $\mathcal{D}$, denoted $\widetilde{s} \models \mathcal{D}$, iff $\widetilde{s} \in Lang(\mathcal{D})$.

Another natural and direct approach to refer to unknown number of agents is to use logical variables: rather than work with atomic propositions p, we can use monadic predicates $p(x)$ to refer to property p being true of client x. We can quantify over such x existentially and universally to specify properties over multiple agents. We are thus naturally led to the realm of Monadic First Order Temporal Logics ($MFOTL$) [35]. Unfortunately, $MFOTL$ is undecidable [45], [70] and we need to limit the expressiveness so that we have decidable verification problem.

Decidable fragments of $MFOTL$ are very few. One of them is the one-variable fragment. Halpern & Vardi[38] and Sistla & German[80] have independently shown that the one-variable fragment of $MFOTL$ with $\Box, \Diamond$ and/or $\mathbf{U}$ is $EXPSPACE$-complete. In [38], the authors consider this fragment as a propositional epistemic temporal logic with one agent modelled by the propositional modal system $S5$. Another decidable fragment of $MFOTL$ is the monodic fragment.

The Monodic Fragment

An $MFOTL$ formula φ is monodic if every subformula $\Diamond\psi, \bigcirc\psi$ or $\psi_1 \mathbf{U} \psi_2$ has at most one free variable, in the scope of ψ's. Decidability is shown by encoding models of a monodic sentence φ in structures called quasimodels and then expressing the statement “there exists a quasimodel satisfying a given monodic sentence” as a monadic second order sentence. Hodkinson *et al.* [44] show that the monodic fragment is also $EXPSPACE$-complete, like the one-variable fragment. In [88], the authors give complete axiomatization for the monodic fragment.

The monodic fragment extended with equality is undecidable [29], even for some very restricted cases, though the “packed” monodic fragment with equality is decidable [43] and, like its pure first-order part [37], is $2EXPSPACE$-complete [44] .

The monodic fragment extended with function symbols is undecidable in general. A single rigid function (a function whose interpretation does not change with time) is sufficient to make the logic not recursively enumerable. However, the monodic monadic fragment with rigid functions, where no two distinct terms have variables bound by the same quantifier, is decidable [50] and $EXPSPACE$-complete.

In [46], the authors analyze the decision problems for fragments of first-order extensions of branching time temporal logics such as CTL and CTL^*. They show that one-variable fragments of logics like first order CTL^* are undecidable. On the other hand, it is proved that by restricting applications of first order quantifiers to *state* (*i.e.*, path independent) formulae, and applications of temporal operators and path quantifiers to formulae with at most one free variable, we can obtain decidable fragments.

Belardinelli & Lomuscio [12] investigate the completeness issues for First Order Epistemic logics. The authors introduce a quantified version of Interpreted Systems (QIS) [11], [10] and show that it can be used to model message passing systems. The class of QIS are used to give semantics to a quantified temporal epistemic logic, called $\mathcal{L}_n$, with no $\bigcirc$ and $\mathbf{U}$ operator. An axiom system $QKT.S5_n$, which is a first-order multi-modal version of $S5$ combined with linear-time temporal logic provides a sound and complete axiomatization for the class QIS.

In [13], the authors identify several monodic fragments of the full first-order temporal logic and prove them to be both sound and complete with respect to the corresponding classes of QISs.

1.4 Contributions of the Thesis

In SCMS systems, the request from a client to a server initiates a sequence of communications between several servers, and eventually an answer is sent to the client. Client-server interactions, as well as server - server interactions are typically temporal in nature, relating to responses received in the past, responses currently awaited, and future actions dependent on various responses. In [67], Meenakshi and Ramanujam propose a local temporal logic (m-LTL) in which such specifications can be written. The systems they study, called Systems of Communicating Automata (SCA), are a variant of CFSMs. The computations of SCAs are a variant of MSCs called Lamport Diagrams, a class of partial orders generated by MSCs. The model checking problem is shown to be decidable, without putting any bound on the channel capacity.

In Chapter 2 of the thesis, we explore the suitability of m-LTL to specify properties of SCMS systems, and find that two changes are appropriate. m-LTL uses an immediate past modality, whereas the (transitive) "some time in the past" modality is found to be more appropriate for services. More interestingly, we advocate the need for a concurrent present modality, called now, to talk about the properties true in the (possibly) present state of some other agent in the system.

The new modality $\langle now \rangle_j$ refers to the present. Over total orders, the present is a point, whereas over partial orders, the present is an interval. $\langle now \rangle_j \beta$, asserted by i, says that β holds in some j-local state concurrent with i's local state. In the thesis, we present a detailed specification of a Travel Agency Web Service illustrating the use of the new modality and distinguish these specifications from those in m-LTL. In Chapter 3 we show that the satisfiability and model checking is decidable for w-LTL.

In the literature [22], SSMC appear in two flavours: discrete and session-oriented. In the case of discrete services, the client sends a request for service to the server and waits for the response, which can either be acceptance or rejection. On the other hand, in the session-oriented paradigm, the client interacts with the server between the send-request and receive-response, in some non-trivial way. This interaction, in turn, may affect the outcome of client request. We can say, that, in the discrete case the clients are passive, whereas, in the other case, the clients are active.

In Chapter 4, we present an automaton model for each paradigm, System of Passive Clients (SPS), for discrete services, and System of Active Clients (SAS), for session-oriented services. In both cases we provide for the facility to admit unbounded number of clients. Yet, the server can remember only a finite number of them at any point of time. Consequently, these are infinite space systems. As a result, their reachability properties are typically undecidable to check. On the other hand, once we bound the number of clients, they reduce to finite state machines. In the same chapter, we show that our models are as rich as standard counter machine models like Vector Addition Systems with States (VASS). Consequently, our models have decidable reachability properties, in general, equivalent to Petri Nets.

There are several candidate temporal logics for message passing systems, but these work with *a priori* fixed number of agents, and for any message, the identity of the sender and the receiver are fixed at design time. In the case of SSMCs, which admit unbounded number of clients, we need to extend such logics with means for referring to agents in some more abstract manner (than by name). A natural and direct approach to refer to unknown clients is to use logical variables, as in $MFOTL$.

In Chapter 6 we present two $MFOTL$ fragments to specify client-server systems with unbounded agents, $\mathcal{L}_{SPS}$ and $\mathcal{L}_{SAS}$ respectively, for SPS and SAS. $\mathcal{L}_{SPS}$ is a combination of LTL and multi-sorted MFO. In the case of LTL, atomic formulae are propositional constants which have no further structure. In $\mathcal{L}_{SPS}$, there are two kind of atomic formulae, basic server properties, and MFO-sentences over client properties. Consequently, these formulae are interpreted over sequences of MFO-structures juxtaposed with LTL-models. We show the satisfiability and model checking of $\mathcal{L}_{SPS}$ to be decidable. The proof uses a formula automaton construction, extending the technique of [84], and in this sense, offers some novelty for a temporal logic with some (limited) quantification.

We propose a fragment of monodic monadic temporal logic ([45]) as the specification language for SAS. In $\mathcal{L}_{SAS}$, the valid specifications hail from the following set:

$$\psi \in \Psi ::= q \in P_s \mid \neg q \mid (\exists x : u)\alpha \mid \psi_1 \vee \psi_2 \mid \psi_1 \wedge \psi_2 \mid \Diamond\psi \mid \Box\psi$$

where α's are client formulae defined as follows:

$$\Delta_x ::= p \in P_c \mid \neg\alpha \mid \alpha_1 \vee \alpha_2 \mid \Diamond\alpha$$

Since it is a fragment of a decidable logic (the monodic $MFOTL$), its satisfiability problem is decidable. In Chapter 6 we present a formula automaton construction, using a multi-counter automaton, that leads to a non-elementary decision procedure for the satisfiability of $\mathcal{L}_{SAS}$.

2

Temporal Specifications of Single Client Multiple Server Systems

Protocols that govern the interaction between software agents in systems of communicating agents [16] form an important and interesting area of study. In the context of single client multiple server systems (SCMS), the specification of how the constituting agents (client and the servers) interact with each other and verifying that models of such systems indeed meet the specification is a central issue. Concurrency plays a central role in this study.

We can conceive of each component in the SCMS as a sequential agent that makes certain choices and seeks information from other agents (or symmetrically, provides information for other agents) in such a way that the SCMS as a whole achieve a desired goal. The computation path of each agent is guided by what it knows about other agents' behaviour, as certified by its interactions with those services.

We follow the premise that concurrency is central to modelling and verification of SCMS, and hence that preserving concurrency information in computations is critical. At the level of requirement specifications, this facilitates local, component-wise reasoning. In terms of models, explicit concurrency (rather than working with interleaved sequences) allows for the study of interaction patterns among agents and communication of state information between them.

To see this, consider the example of a set of services together managing the travel requirements of a client. (We will see more of this example later.) There is one agent handling airline reservation, one exploring train reservation and another handling hotel booking. The choices made by the hotel agent crucially depend on the choices made by the other two and this coordination is required *within a computation*. This is easily modelled using partial ordering of events in a computation: a coordinated choice corresponds to one concurrent computation, where events of each service are linearly ordered, whereas globally the order is partial, with concurrent events being incomparable. This facilitates references (or assumptions) on the hotel service's part to what is concurrently possibly taking place at the train reservation service.

With such a motivation we define a partial order based model of computations of SCMS called Lamport diagrams and a local temporal logic to describe the set of good (and/or bad) computations.

2.1 Lamport Diagrams

We know that behaviours of sequential systems can be described by finite or infinite words over a suitable alphabet of actions. The system has an underlying set of events and an alphabet of actions that label the event occurrences. A word over such an alphabet represents a behaviour of the system as a totally ordered sequence of actions of the system and a set of such words represents possible behaviours of the system. Extending this intuition, Lamport [60] suggested that we could use partial orders to represent computations of concurrent systems. Since event occurrences of different agents can be independent of each other, events of the system are partially ordered. Lamport diagrams, representing non-sequential runs, are partial orders with the underlying set of events partitioned into those of n agents in such a way that the event occurrences of every agent form a linear order. The ordering relation captures the causal dependence of event occurrences. Lamport Diagrams were first defined formally in [67], the discussion here is taken from [66]. We fix the number of agents in the system as n and take $[n] = \{1, 2, \cdots, n\}$. A distributed alphabet for such systems is an n-tuple $\widetilde{\Sigma} = (\Sigma_1, \ldots, \Sigma_n)$, where for each $i \in [n]$, Σ_i is a finite non-empty alphabet of actions of agent i and for all $i \neq j$, $\Sigma_i \cap \Sigma_j = \emptyset$. The alphabet induced by $\widetilde{\Sigma} = (\Sigma_1, \ldots, \Sigma_n)$ is given by $\Sigma = \bigcup_i \Sigma_i$. We define ($\Sigma$-labelled) Lamport diagrams formally, as follows:

Definition 2.1.1. *A* **Lamport diagram** *is a tuple $D = (E, \leq, V)$ where*

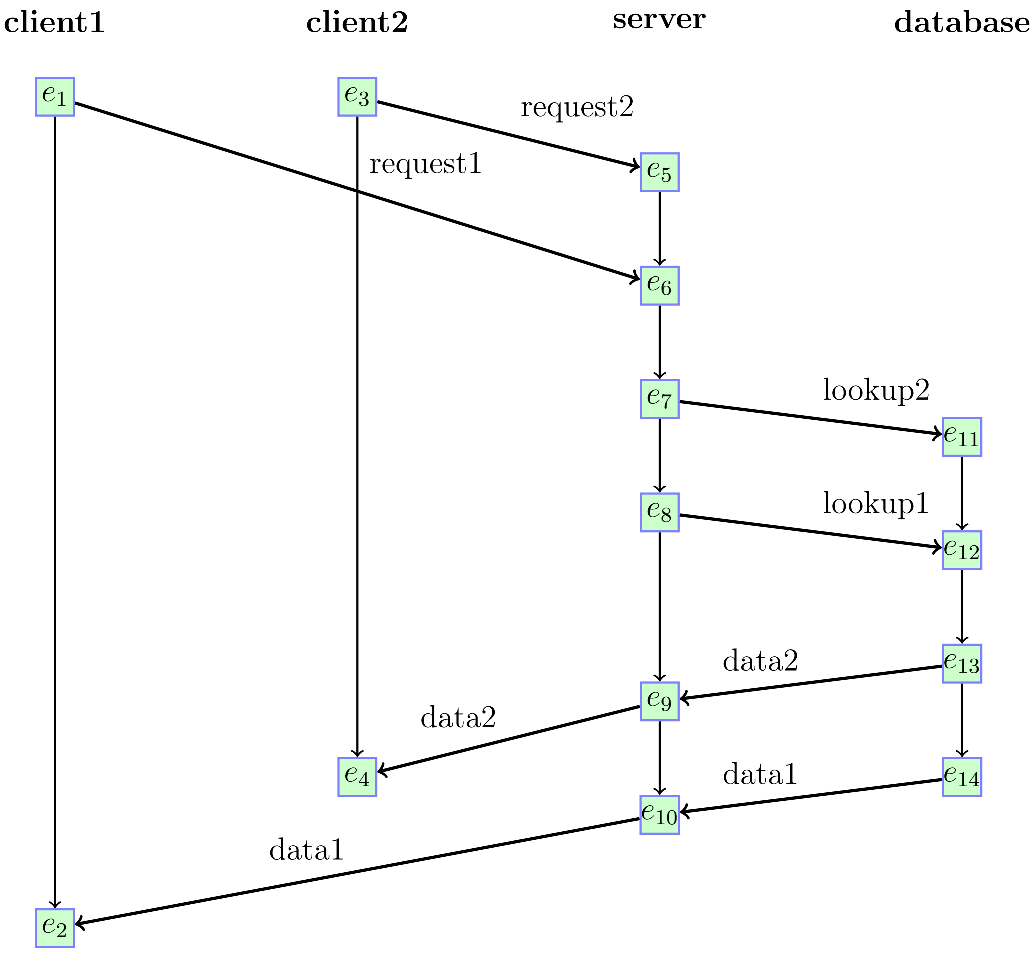


Figure 2.1: Lamport diagram representing a scenario of client-server system

- *E is an at most countable set of* events.
- *$\leq \subseteq (E \times E)$ is a partial order called the* causality relation *such that for every $e \in E$, $\downarrow e \overset{\text{def}}{=} \{e' \in E \mid e' \leq e\}$ is finite.*
- *$V : E \rightarrow \Sigma$ is a* labelling function *which satisfies the following condition:*

  Let $E_i \overset{\text{def}}{=} \{e \in E \mid V(e) \in \Sigma_i\}$ and $\leq_i \overset{\text{def}}{=} \leq \cap (E_i \times E_i)$. then for every $i \in [n]$, $\leq_i$ is a total order on E_i.

In the above definition, the relation $\leq$ describes the causal dependence of events and the relations $\{\leq_i \mid i \in [n]\}$ capture the fact that event occurrences of each agent are totally ordered. Note that the labelling function V implicitly assigns a unique agent to every event. For example, if $V(e) \in \Sigma_i$ for some $e \in E$ and $i \in [n]$, then the event e belongs to the agent i. This, in turn, rules out any synchronous communication in the underlying system, as, in that case, a synchronous or joint communication event occurrence would be associated with more than one agent.

For example, consider a system where a fixed finite set of clients are registered with a server that provides them access to a database. A Lamport diagram representing a behaviour of the system is depicted in Figure 2.1. There are four agents: client1 and client2 are two clients registered with the server in order to access the database. Event e_1 is an event occurrence of the agent client1 corresponding to sending of the message request1 to the server. The receipt of this message by the server is represented by the event occurrence e_6. The server passes the requests to the database (represented by lookup1 and lookup2) and the response from the database (data1 and data2) is communicated back to the clients. Observe that event occurrences e_1 and e_3 corresponding to sending of requests from client1 and client2 respectively are *concurrent*, *i.e.*, they are not causally dependent on each other. On the other hand, the event occurrences e_5 and e_6 corresponding to the receipt of the messages request1 and request2 respectively, are causally dependent. Thus, request1 and request2 are concurrently originating but sequentialized by the server computation. Similarly, for e_{13} to occur, e_7 and e_1 should have already occurred. It is in this sense that Lamport diagrams depict the causal dependence of various event occurrences within a system computation.

To be precise, the relation $\leq$ is causal in the sense that whenever $e \leq e'$, we interpret this as the condition that, in any run of the system, e' cannot occur without e having occurred previously in that run. Since for all $e \in E$, $\downarrow e$ is finite, $\leq$ must be discrete. Hence there exists $\lessdot \subset \leq$, the immediate causality relation, which generates the causality relation; that is: for all e, e', $e \lessdot e'$ iff $e < e'$ and for all $e'' \in E$ if $e \leq e'' \leq e'$ then either $e'' = e$ or $e'' = e'$. We have $\leq = (\lessdot)^*$. Now consider $e \lessdot e'$. If $e, e' \in E_i$ for some $i \in [n]$, we see this as local causal dependence. However, if $e \in E_i$ and $e' \in E_j$, $i, j \in [n]$, $i \neq j$, we have remote causal dependence. For $e, e' \in E$, define $e <_c e'$ iff $e \in E_i$, $e' \in E_j$, $i \neq j$ and $e \lessdot e'$. In this case, we interpret e as the sending of a message by agent i and e' as its corresponding receipt by j. Accordingly, if $e <_c e'$ then e will be referred to as a send event and e' will be its corresponding receive event. An event e will be interpreted as a local event if there exists no e' such that $e <_c e'$ or $e' <_c e$. Notice that the communication relation $<_c$ is derived from the Hasse diagram of the causal dependence relation which is a partial order.

Note that given an event $e \in E$, there can be at most n events e' such that $e \lessdot e'$ and at most n events e' such that $e' \lessdot e$. In particular, if $e \in E_i$ and $e <_c e'$, $e <_c e''$ where $e' \in E_j$ and $e'' \in E_k$ for $j, k \in [n]$ such that $j \neq i$ and $k \neq i$, then e is a send event simultaneously to agents j and k. Such events can be thought of as representing broadcast type of communication where a common message is broadcast to several agents in the system. Similarly, there can be events which are simultaneous receive events from more than one agent. Also, an event e can be a send and a receive event simultaneously. For example, the events e_9 and e_{10} in the Lamport diagram given in Figure 2.1 are events which represent send and receive actions simultaneously.

2.1.1 States of a Lamport Diagram

The concept of global state in a Lamport diagram is given by the notion of a configuration, which is any downward closed set of events. That is, $c \subseteq E$ is a configuration iff for all $e \in c$, $\downarrow e \subseteq c$. For example, the set $c_1 = \{e_1, e_3, e_5, e_6\}$ is a configuration of the Lamport diagram given in Figure 2.1 whereas $c_2 = \{e_6, e_7\}$ is not a configuration as $e_1 \in \downarrow e_6$ but $e_1 \notin c_2$. Given a Lamport diagram D, let C_D^{fin} denote the set of all finite configurations of D. The empty configuration corresponds to the initial global state when no event has occurred and is denoted by $\emptyset$.

For each $i \in [n]$ and any finite configuration c, if $c \cap E_i \neq \emptyset$, then there exists $e_i \in c \cap E_i$ which is the maximum with respect to $\leq$ (as $\leq_i$ is a total order on E_i). Hence, a finite configuration c can be represented by an n-tuple $(x_1, x_2, \ldots, x_n)$ where for $i \in [n]$, $x_i = e_i$ iff $c \cap E_i \neq \emptyset$ and e_i is the maximum event of E_i in c and $x_i = \perp$ otherwise. c is then given by $c = \cup_{i=1}^{n} \downarrow x_i$ where $\downarrow \perp = \emptyset$. For example, the configuration c_1 in the Lamport diagram of Figure 2.1 can be represented by the tuple $(e_1, e_3, e_6, \perp)$.

Let $e \in E_i$. Note that $\downarrow e$ is a configuration, and we can think of $\downarrow e$ as the local state of agent i when the event e has just occurred. This state contains the information that i has up till that instant in the computation, which contains its own local history and that of other agents according to the latest communication from them. The empty set corresponds to the local initial state, where no i-event has occurred, and is denoted by ϵ_i. We shall use $\emptyset$ interchangeably with ϵ_i while referring to initial state of agent i. Let the set of all local states of agent i be denoted $LC_i \overset{\text{def}}{=} \{\epsilon_i\} \cup \{{\downarrow}e \mid e \in E_i\}$ and let $LC \overset{\text{def}}{=} \bigcup\limits_i LC_i$. We use d, d', d_1 etc to denote local states. It is trivially seen that we can extend $\leq$ to the set of all local states. For all $d, d' \in LC$, $d \leq d'$ if $d \subseteq d'$. We can also extend the $\lessdot$ relation to local states as follows: let $d_1 \in LC_j$ and $d_2 \in LC_i$; we say $d_1 \lessdot d_2$ iff $d_1 = d_2 = \emptyset$ or $d_1 \subset d_2$ and for all $d \in LC_j$, if $d \subseteq d_2$, then $d \subseteq d_1$ as well; that is, d_1 is the last j-local state seen by i at d_2. Therefore, $\epsilon_j \lessdot \epsilon_i$ for all $j \neq i$.

Given a Lamport diagram $D = (E, \leq, V)$, we can have a labelling defined over local states of D. $\widehat{V} : LC \to \Sigma$ is a valid extension of V if the following hold:

- for each $e \in E$, $\widehat{V}({\downarrow}e) = V(e)$ and
- for each $i \in [n]$, $\widehat{V}(\epsilon_i) \in \Sigma_i$.

We denote $\widehat{V}$ by V, when there is no confusion.

Let $d \in LC_i, d' \in LC_j, j \neq i$. d and d' are said to be compatible if there exists a global configuration in which d is the most recent i-local state and d' is the most recent j-local state. In this situation, agent i can consider, at d, that j could *at present* be at d'. This is formalized by the following relation: $\rangle\langle \subseteq (LC \times LC)$ is defined by:

$$\{(d, d') \mid d \in LC_i, d' \in LC_j, j \neq i, \exists(x_1, x_2, \ldots, x_n) \in C_D^{fin} : {\downarrow}x_i = d, {\downarrow}x_j = d'\}.$$

2.2 The Logic *wm*-LTL

The logical language which we use to specify single client multiple server systems (SCMS) is a local temporal logic. A local temporal logic is similar to the standard linear time temporal logic for multiple agents, but, here assertions are made at local states of component agents and such assertions can refer to what may be concurrently occurring at other agents. These formulas are interpreted on Lamport diagrams. The SCMSs are modelled by communicating finite state automata (SCAs, more on them in the next chapter) and the behaviours of such systems are given by collections of Lamport diagrams. This allows us to define the model checking problem for the logic and prove it to be decidable. The formal issues regarding the logic in question are tackled in the next chapter.

The logic presented here is related to m-LTL ([67]) which studied temporal logics over Lamport diagrams. However the use of the $\langle now \rangle$ modality for concurrency is new and forms an important motivation here. The $\langle now \rangle$ modality was first discussed in [75] but in the context of other partial orders (Mazurkiewicz traces) where the technical machinery involved is very different.

2.2.1 Syntax and Semantics

Fix countable sets of *propositional letters* $(P_1, P_2, \ldots, P_n)$, where P_i consists of the atomic local properties of agent i. We assume, for convenience, that $P_i \cap P_j = \emptyset$ for $i \neq j$. Let $P \stackrel{\text{def}}{=} \bigcup\limits_i P_i$.

Let $i \in [n]$. The syntax of *i-local formulas* is given below:

$$\Phi_i ::= p \ \in \ P_i \mid \neg \ \alpha \mid \alpha_1 \ \vee \ \alpha_2 \mid \bigcirc \ \alpha \mid \alpha_1 \ \mathbf{U} \ \alpha_2 \mid \diamondminus \ \alpha \mid \langle now \rangle_j \beta \mid \ominus_j \beta$$

Above, $j \neq i$, $\alpha \in \Phi_i$ and $\beta \ \in \ \Phi_j$.

Global formulas are obtained by boolean combination of local formulas:

$$\Psi ::= \alpha @ i, \ \alpha \ \in \ \Phi_i \mid \neg \ \psi \mid \psi_1 \ \vee \ \psi_2$$

The propositional connectives $(\wedge, \supset, \equiv, \oplus)$ and derived temporal modalities $(\Diamond, \Box)$ are defined as usual. In particular, $\Diamond\alpha = T\mathbf{U}\alpha$ and $\Box\alpha \equiv \neg\Diamond\neg\alpha$. The dual of $\langle now \rangle_j \beta$ is given by $[now]_j \beta \stackrel{\text{def}}{=} \neg\langle now \rangle_j \neg\beta$.

Note that $\diamondminus$ is a local modality whereas $\langle now \rangle_j$ and $\ominus_j$ are non-local. We can fashion a non-local past modality using $\langle now \rangle_j$ as follows:

$$\diamondminus_j \beta \equiv [now]_j \diamondminus \alpha.$$

The dual of $\diamondminus_j \beta$ is given by $\boxminus_j \beta \stackrel{\text{def}}{=} \neg\diamondminus_j \neg\beta$.

The formulas are interpreted on Lamport diagrams. For technical convenience, we consider only infinite behaviours. Formally, models are 2^P-labelled Lamport diagrams over a countable set of events. $M = (E, \leq, V)$ is a Lamport diagram such that E is countably infinite and $V : LC \rightarrow 2^P$ such that for all $d \in LC_i$,

$V(d) \subseteq P_i$.

The labelling of the local states of D by subsets of P can be thought of as associating a valuation function with the Lamport diagram. We choose to consider the valuation function as a label as it would be easier to associate an SCA with every formula later.

Let $\alpha \in \Phi_i$ and $d \in LC_i$. The notion that α *holds in the local state* d of agent i in model M is denoted $M, d \models_i \alpha$, and is defined inductively as follows:

- $M, d \models_i p$ iff $p \in V(d)$.
- $M, d \models_i \neg\alpha$ iff $M, d \not\models_i \alpha$.
- $M, d \models_i \alpha \vee \alpha'$ iff $M, d \models_i \alpha$ or $M, d \models_i \alpha'$.
- $M, d \models_i \bigcirc\alpha$ iff there exists $d' \in LC_i$ such that $d \lessdot d'$ and $M, d' \models_i \alpha$.
- $M, d \models_i \alpha\mathbf{U}\alpha'$ iff $\exists d' \in LC_i$: $d \leq d', M, d' \models_i \alpha'$ and $\forall d'' \in LC_i : d \leq d'' < d' : M, d'' \models_i \alpha$.
- $M, d \models_i \diamondminus\alpha$ iff there exists $d' \in LC_i$ such that $d' \leq d$ and $M, d' \models_i \alpha$.
- $M, d \models_i \diamondminus_j\beta$ iff there exists $d' \in LC_j$ such that $d' \leq d$ and $M, d' \models_j \beta$.
- $M, d \models_i \langle now \rangle_j \beta$ iff there exists $d' \in LC_j$ such that $d' \rangle\langle d$ and $M, d' \models_j \beta$.
- $M, d \models_i \ominus_j\beta$ iff there exists $d' \in LC_j$ such that $d' \lessdot d$ and $M, d' \models_j \beta$.

The modalities $\bigcirc$ (next) and $\mathbf{U}$ (until) are standard linear time temporal logic operators interpreted at the local future of a state. The strongly global modality $\ominus_j\beta$, asserted by i, says that β held in the most recent past j-local state visible to i. The weakly global modality $\diamondminus_j\beta$, asserted by i, says that β held in some past j-local state visible to i. Notice that this need not be through a 'direct edge' from j to i, *i.e.*, there need not be any communication between i and j directly. The same is true for $\ominus_j\beta$.

The modality $\langle now \rangle_j$ refers to the present. Over total orders, the present is a point, whereas over partial orders, the present is an interval. $\langle now \rangle_j\beta$, asserted by i, says that β holds in some j-local state concurrent with i's local state.

$\langle now \rangle_j$ modality is particularly useful when there is no communication between agent i and agent j. In such a scenario agent i, in local state d, can guess the states d' compatible with d and accordingly make its moves.

On the other hand, $\ominus_j$ is useful when there is a flow of information between agents i and j, either directly or indirectly, via message passing. In such a scenario agent i can assert what it knows about the agent j and accordingly make its moves.

Global satisfaction is defined in terms of local satisfaction at the initial events. There are other models where global satisfaction may be defined in terms of local satisfaction at arbitrary events, but that makes the logic undecidable.

- $M \models \alpha@i$ iff $M, \epsilon_i \models_i \alpha$.
- $M \models \neg\psi$ iff $M \not\models \psi$.
- $M \models \psi_1 \vee \psi_2$ iff $M \models \psi_1$ or $M \models \psi_2$.

We say that ψ is *satisfiable* iff there exists a model M such that $M \models \psi$. We say that ψ is *valid* if for every model M, we have $M \models \psi$. Let $\mathit{Models}(\psi) = \{M \mid M \models \psi\}$.

2.3 Bisimulation Invariance of *wm*-LTL

We recall the definitions of bisimulation and bisimulation invariance in the context of Lamport diagrams. Given a pair of Lamport diagrams $D_1 = (E_1, \leq_1, V_1)$ and $D_2 = (E_2, \leq_2, V_2)$, over n agents, we are interested in whether or not two models (labelled Lamport diagrams) are equivalent under properties expressible in *wm*-LTL.

Definition 2.3.1. *Given two n-agent Lamport diagrams over distributed alphabet $\widetilde{\Sigma}$, $D_1 = (E_1, \leq_1, V_1)$ and $D_2 = (E_2, \leq_2, V_2)$, a non-empty relation $\mathbf{R} \subseteq LC^1 \times LC^2$ is a* bisimulation *if for every $j \in [n]$, for every $d_1 \in LC^1_j, d_2 \in LC^2_j$, $d_1\mathbf{R}d_2$ iff the following conditions hold:*

(valuation) *for every $p \in P$, $p \in V_1(d_1)$ iff $p \in V_2(d_2)$;*

(past-forth) *for every $j' \in [n]$, for every $d'_1 \in LC^1_{j'}$ if $d'_1 \lessdot d_1$ then there exists $d'_2 \in LC^2_{j'}$ such that $d'_2 \lessdot d_2$ and $d'_1\mathbf{R}d'_2$;*

(past-back) *for every* $j' \in [n]$, *for every* $d_2' \in LC^2_{j'}$ *if* $d_2' \lessdot d_2$ *then there exists* $d_1' \in LC^1_{j'}$ *such that* $d_1' \lessdot d_1$ *and* $d_1'\mathbf{R}d_2'$;

(future-forth) *for every* $j' \in [n]$, *for every* $d_1' \in LC^1_{j'}$ *if* $d_1 \lessdot d_1'$ *then there exists* $d_2' \in LC^2_{j'}$ *such that* $d_2 \lessdot d_2'$ *and* $d_1'\mathbf{R}d_2'$;

(future-back) *for every* $j' \in [n]$, *for every* $d_2' \in LC^1_{j'}$ *if* $d_2 \lessdot d_2'$ *then there exists* $d_1' \in LC^2_{j'}$ *such that* $d_1 \lessdot d_1'$ *and* $d_1'\mathbf{R}d_2'$.

(conc-forth) *for every* $j' \in [n]$, *for every* $d_1' \in LC^1_{j'}$ *such that* $d_1\rangle\langle d_1'$ *there exists* $d_2' \in LC^2_{j'}$ *such that* $d_2\rangle\langle d_2'$ *and* $d_1'\mathbf{R}d_2'$;

(conc-back) *for every* $j' \in [n]$, *for every* $d_2' \in LC^2_{j'}$ *such that* $d_2\rangle\langle d_2'$ *there exists* $d_1' \in LC^1_{j'}$ *such that* $d_1\rangle\langle d_1'$ *and* $d_1'\mathbf{R}d_2'$;

We say D_1 ***bisimilar*** D_2, *i.e.,* $D_1 \sim D_2$, *if there exists a bisimulation* $\mathbf{R}$ *such that for every* $j \in [n]$, $\varepsilon_j\mathbf{R}\varepsilon_j'$.

Definition 2.3.2. *A property* ψ *is* bisimulation invariant *if the following holds:*

$$\textit{if } D_1 \sim D_2 \textit{ and } D_1 \models \psi \textit{ then } D_2 \models \psi.$$

We can now show that the logic *wm*-LTL is bisimulation invariant.

Lemma 2.3.3. *Suppose* $D_1 \sim D_2$ *with a bisimulation* $\mathbf{R}$. *Then for every* $j \in [n]$, *for every* $d_1 \in LC^1_j, d_2 \in LC'_j$ *such that* $d_1\mathbf{R}d_2$, *for every* $\alpha \in \Phi_j$, $D_1, d_1 \models_j \alpha$ *iff* $D_2, d_2 \models_j \alpha$.

The proof is by induction on the structure of α and follows easily from the semantics of modalities and the definition 2.3.1. Now, for a given $\psi \in \Psi$ in *wm*-LTL, by induction on the structure of ψ we can assert the following:

Theorem 2.3.4. *Given two n-agent Lamport diagrams over distributed alphabet* $\widetilde{\Sigma}$, $D_1 = (E_1, \leq_1, V_1)$ *and* $D_2 = (E_2, \leq_2, V_2)$, *for every wm-LTL formula* $\psi \in \Psi$, *if* $D_1 \sim D_2$ *then* $D_1 \models \psi$ *iff* $D_2 \models \psi$.

2.4 Specification Examples

The specification examples describe the working of Travel Agency Web Service. This composite system consists of four agents or types of agents; customer, travel agent, service providers viz train reservation, airline reservation, hotel accommodation etc and credit card companies.

The composite system works as follows: The customer (c) contacts the travel agent (ta) to fix her travel plans for a given destination, with travel schedule, duration of stay, choice of routes, flexibility in dates and budget constraints. The travel agent communicates with train reservation (t) and airline reservation (a) checking fares and availability, and ensuring that accommodation is available by checking with the hotel accommodation service. After one or more tentative plans are drawn up, the travel agent gets back to the customer for her choice. The credit card company enables customers to use their credit cards. Typically, only the customer in the scenario may be a human being and the travel agent service, airline, hotel and payment services that the travel agent service interacts with are programs. [1]

There can be many scenarios of interaction of travel agent ta with service providers (t, a and h). We distinguish two particular cases. In the first case, which we call sim_req, as this proposition is asserted locally in ta, ta sends simultaneous requests to service providers t, a and h. Thus, when each service provider comes to fix what service/s it is going to offer it has no idea what others are up to. In the other case, which we call lin_req, ta sends the requests to service providers in some non-simultaneous fashion. Here it is probable that one agent knows what another has offered owing to the sequentiality of requests. That is, in the sim_req case, each agent can only assert (in fact, guess) facts about other agents using $\langle now \rangle$ modality and decide its own offers, whereas, in the lin_req case, each agent can obtain information using $\ominus$ and decide.

At the outset, we enumerate the atomic local properties of individual agents, that is the P_i's.

- $P_c = \{req\_pending, more\_options, low\_end\}$ are the local properties of

[1] The above description is taken from "http://www.w3.org/2002/06/ws-example" titled "Web service use case:Travel reservation". A real life example can be found at "www.webtravelservices.co.uk".

the customer. $req\_pending$ means a request to the travel agent has been sent and the customer is waiting to hear from her. $more\_options$ means the customer has got a few options from the travel agent about the holiday package but she would like to see others too before making a call. When low_end holds, it means the customer expects a cheaper travel package. $high\_end \equiv \neg low\_end$ means the customer may well do with a costlier package.

- $P_{ta} = \{req\_train, req\_air, lin\_req, sim\_req, waiting, tr\_rq\_disable, air\_rq\_disable\}$ are the local properties of the travel agent. req_train means a request has been sent to train service for offers, whereas req_air means a request has been sent to airline service. lin_req holds when the requests to hotel and airline service are sent in order, one way or the other whereas sim_req holds when the requests are sent simultaneously. $waiting$ holds when requests have been sent and the travel agent is waiting for the respective offers from the services. $tr\_rq\_disable, air\_rq\_disable$ hold, respectively, when travel agent no longer wants any service from train and airline service.

- $P_t = \{ac, cnfd, 2sl\}$ are the local properties of the train reservation service. ac holds when the train service offers berths with AC accommodation, $2sl$ holds when the berths are non-AC (*i.e.*, second class sleeper) and $cnfd$ when the bookings are confirmed either AC or non-AC.

- $P_a = \{direct\_flight, high\_fare\}$ are the local properties of the airline reservation service. $direct\_flight$ holds when the airline service offers bookings on direct fights to the destination whereas $high\_fare$ denote bookings with high fare.

- $P_h = \{off\_season, rooms, cottages\}$ are the local properties of the hotel accommodation service. off_season holds when hotel accommodation service offers rooms and cottages at off-season (low) rates, $rooms$ holds when rooms are available and $cottages$ when high-end cottages are available.

Using the logical language *wm*-LTL, in particular the $\ominus$ modality, we can say many interesting things as follows:

First we look at specifications for the hotel accommodation service. In the case where travel agent has made simultaneous requests the hotel service has no idea what other service providers, airline and train, are going to offer, so it offers off-season rates for budget customer. Therefore, we can have specifications of the kind if the travel agent has made simultaneous request then hotel offers off-season rates which is formally written as

$$\ominus_{ta} sim\_req \supset \Diamond off\_season.$$

On the other hand, if the travel agent has made sequential requests and if airline offers high fares or train offers AC tickets then hotel offers cottages. This is written as follows:

$$\ominus_{ta} lin\_req \supset (\Diamond(\ominus_a high\_fare \vee \ominus_t ac) \supset \Diamond cottages)$$

Next, we look at specifications for the airline reservation service. The default specification for airline reservation service, when the travel agent has made simultaneous requests, is if the travel agent has made simultaneous request then airline offers high fare direct flights. In *wm*-LTL,

$$\ominus_{ta} sim\_req \supset \Box(high\_fare \wedge dir\_flight).$$

In the other situation, when there are sequential requests, we have if travel agent has sent linear requests then if train is known to offer AC tickets then airline offers low-fare flights. This is written as

$$\ominus_{ta} lin\_req \supset \Box(\ominus_t ac \supset \Diamond low\_fare).$$

Furthermore, if travel agent has sent linear requests then if train is known to offer non-confirmed tickets then airline offers direct flights can be framed as

$$\ominus_{ta} lin\_req \supset \Box(\ominus_t waiting \supset \Diamond dir\_flight).$$

Note that the specifications for airline are fashioned in the aforementioned way as it is competing for customers with train service. The default specification for train reservation service is if the travel agent has made simultaneous request then train offers confirmed AC tickets, which is framed as follows in *wm*-LTL:

$$\ominus_{ta} sim\_req \supset \Box(cnfd \wedge ac).$$

In the case of sequential requests, the specification could be if travel agent has sent linear requests and if airline is known to offer high-fare tickets then train offers confirmed non-AC berths framed as

$$\ominus_{ta} lin\_req \supset \Box(\ominus_a high\_fare \supset \Box(cnfd \wedge 2sl))$$

and if travel agent has sent linear requests and if airline is known to offer non-direct flights then train service offers confirmed tickets framed as

$$\ominus_{ta} lin\_req \supset \Box(\ominus_t no\_dir\_flight \supset \Box cnfd).$$

Note that the specifications for train service are geared in such a way that it always offers confirmed tickets, whether AC or non-AC, as it is constantly at a disadvantage competing with airline.

Travel Agent simply passes on the offers from service providers to the customer, waits for confirmation and, may be requests for more options from them if demanded by the customer. We would like the travel agent to follow specifications of the type at any time, if the offer from train service is non-AC non-confirmed then travel agent stops sending requests to it and at any time, if the offer from airline service is high-fare indirect flights then travel agent stops sending requests to it. In *wm*-LTL, they are framed, respectively, as

$$\Box(\ominus_t(waiting \wedge 2sl) \supset \bigcirc\Box tr\_rq\_disable) \text{ and}$$

$$\Box(\ominus_a(high\_fare \wedge no\_dir\_flight) \supset \bigcirc\Box air\_rq\_disable).$$

We conclude with specifications for the customer. if i'm a low-end customer then i shall accept a package only when the offer is non-AC confirmed from train service or cheap flights from airline coupled with rooms on off-season rates framed in *wm*-LTL as follows:

$$\Box(low\_end \supset ((\ominus_a low\_fare \vee \ominus_t(2sl \wedge cnfd)) \wedge \ominus_h(rooms \wedge off\_season) \supset acc)$$

and if i'm a high-end customer then i shall accept any package with confirmed train tickets and/or direct flights framed as follows:

$$\Box(high\_end \supset (\ominus_a dir\_flight \vee \ominus_t cnfd) \supset acc).$$

Using the $\langle now \rangle$ modality, we can say many *other* interesting things. In particular, when a travel agent has made simultaneous requests, a service provider can refer to the services offered by the other two, using the now modality, and make a decision for itself. For example, the hotel accommodation service can be specified as if the travel agent is known to have made simultaneous request then hotel decides to offer only cottages if the train is expected to offer AC and airline high-fare flights. This is written as follows in *wm*-LTL:

$$\Box((\diamondminus_{ta} sim\_req \wedge \langle now \rangle_t ac \wedge \langle now \rangle_a high\_fare) \supset \Diamond cottages).$$

For airline reservation service the specification could be if the travel agent has made simultaneous request then airline offers low fare flights expecting the train service to offer AC tickets framed in *wm*-LTL as

$$\Box((\diamondminus_{ta} sim\_req \wedge \langle now \rangle_t ac) \supset \Diamond low\_fare).$$

Whereas, for train reservation service we have the following: if the travel agent has made simultaneous request then train offers confirmed AC berths, expecting airline to offer high-fare tickets which is expressed as follows in *wm*-LTL:

$$\Box((\diamondminus_{ta} sim\_req \wedge \langle now \rangle_a high\_fare) \supset \Diamond(cnfd \wedge 2sl)).$$

The global specification for the composite system is:

$$\psi = \alpha_c @ c \wedge \alpha_{ta} @ ta \wedge \alpha_t @ t \wedge \alpha_a @ a \wedge \alpha_h @ h$$

where α_c is the conjunction of customer specifications given above, α_{ta} is the conjunction of specifications for travel agent given above, and so on.

2.4.1 Specifications for Quote-request Web service

As already mentioned, the quote-request Web service consists of a buyer client and two types of servers, suppliers and manufacturers (producers), each with possibly multiple instances. The buyer interacts with the multiple suppliers in order to purchase some commodity, say cars or obtain some service, health insurance for employees. The suppliers, in turn, interact with the multiple manufacturers and prepare a quote each for the buyer.

The basic steps of interaction are as follows: The buyer requests a quote from a set of suppliers. All suppliers receive the request for quote and send request for bills of items to their respective manufacturers. The suppliers interact with their manufacturers to build their quotes for the buyer. The eventual quote is then sent to the buyer. When the buyer receives the quotes one of the following may occur: The buyer agrees with one or more quotes and places the order (or orders), and terminates. Otherwise, the buyer rejects quotes and terminates. [2]

We can describe many interesting properties of such pattern of interactions among buyer B, suppliers and manufacturers using *wm*-LTL. For simplicity, we assume there are two suppliers, S_1, S_2 each with its captive manufacturer, M_1, M_2, respectively. To begin with, we describe the set of local properties attached to each agent.

1. P_b, the set of local propositions for buyer with their expected meanings is enumerated as follows:

   (a) snd_quote_req,

   (b) $rcv\_Q1\_i, rcv\_Q2\_i, rcv\_Q3\_i$, $i = 1, 2$, where $Q1, Q2, Q3$ are quotes, that are discrete values and $Q1 < Q2 < Q3$,

   (c) $acc\_Q1\_i, acc\_Q2\_i, acc\_Q3\_i$, $i = 1, 2$ and

   (d) $rej\_Q1\_i, rej\_Q2\_i, rej\_Q3\_i$, $i = 1, 2$.

2. P_s^i, $i = 1, 2$, is the set of local propositions for supplier i with their expected meanings is enumerated as follows:

   (a) $rcv\_quote\_req\_i$,

[2] The above description is taken from "http://www.w3.org/TR/ws-chor-reqs/" titled "Web service Choreography Requirements".

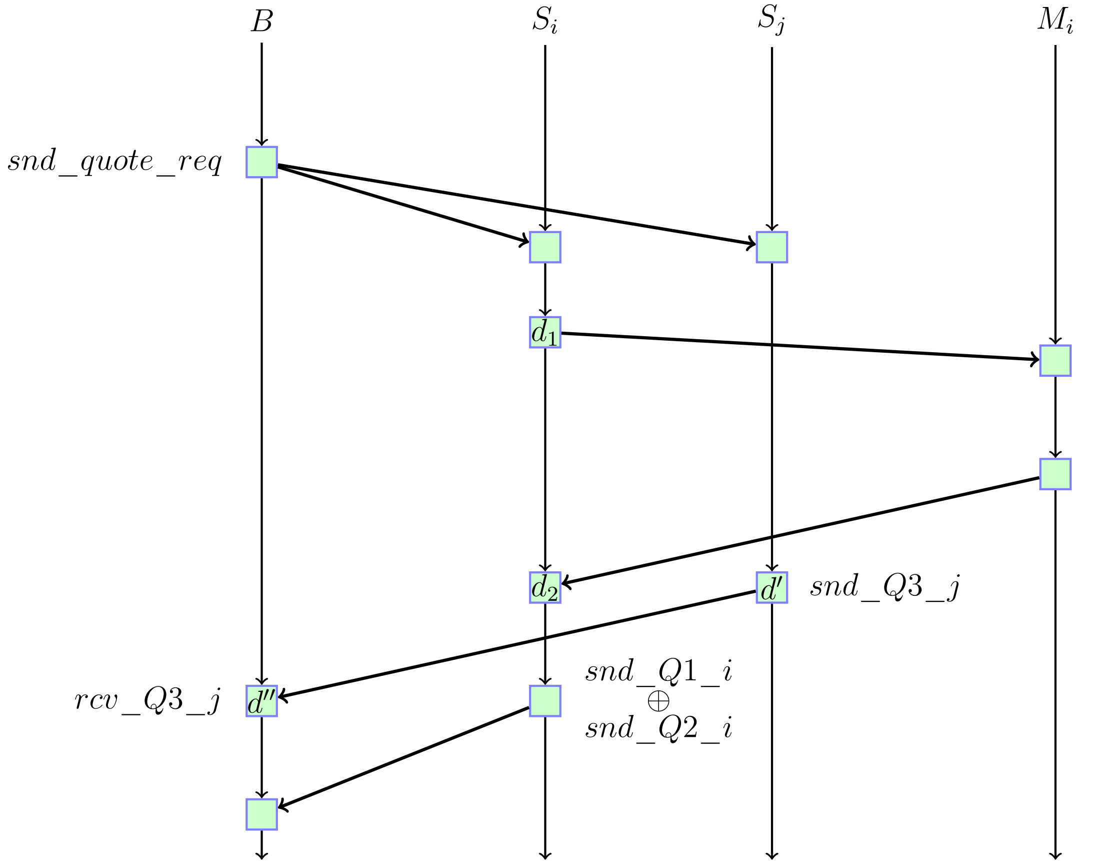


Figure 2.2: Fragment of a quote-request WS execution pattern showing compatible local states $d_2\rangle\langle d'$ and $d_2\rangle\langle d'$

   (b) $snd\_bill\_req\_i$,

   (c) $rcv\_B1\_i, rcv\_B2\_i, rcv\_B3\_i$, where $B1, B2, B3$ are bills, that are discrete values and $B1 < B2 < B3$ and

   (d) $snd\_Q1\_i, snd\_Q2\_i, snd\_Q3\_i$.

3. P_m^i, $i = 1, 2$, the local propositions of manufacturer i with their expected meanings is enumerated as follows:

   (a) $rcv\_bill\_req\_i$ and

   (b) $snd\_B1\_i, snd\_B2\_i, snd\_B3\_i$.

First we mention those specifications which use $\langle now \rangle$ modality. These specifications can not be expressed using $\ominus$ modality. There are two scenarios where $\langle now \rangle$ modality can be used as shown in the Figure 2.2.

In the first case, the supplier S_1 can guess the states in S_2 as there is no communication between them either directly or indirectly and modify its quote. The following formulae can, therefore, be asserted in S_1.

1. if S_2 has sent a quote $Q3$ then i shall send the quote $Q1$ or $Q2$,

$$\Box\big(\langle now \rangle_{S_2} snd\_Q3\_2 \supset \Diamond(snd\_Q1\_1 \oplus snd\_Q2\_1)\big),$$

2. if S_2 has received a bill B_3 then i shall send the quote $Q1$ or $Q2$,

$$\Box\big(\langle now \rangle_{S_2} rcv\_B3\_2 \supset \Diamond(snd\_Q1\_1 \oplus snd\_Q2\_1)\big).$$

Also, S_1 can guess the states in buyer between the states asserting snd_quote_req and rcv_Qk_1, respectively, and modify its quote. The following formula can, therefore, be asserted in S_1 which means, if the buyer has received the quote of $Q3$ then i shall either send the quote $Q1$ or $Q2$,

$$\Box\big(\langle now \rangle_{B} rcv\_Q3\_2 \supset \Diamond(snd\_Q1\_i \oplus snd\_Q2\_1)\big).$$

In the case of specifications of manufacturers too, we can use $\langle now \rangle$ to guess what other manufacturers are preparing for bill. This is non-trivial only when a supplier does not have a dedicated manufacturer. We can take into account competition among manufacturers and add more specifications. We do not explore such specifications here.

The $\ominus$ modality is particularly useful when there is significant communication between the agents of the system. Consider the quote-request system with a few modifications. In this case, apart from the buyer B, there is only one supplier S which interacts with multiple competing manufacturers and prepares a quote for the buyer. For simplicity, we assume only two manufacturers in the system, M_1 and M_2.

After a request-for-quote from the buyer the supplier asks for bills from the manufacturers, one after the another. Although there is no direct communication between M_1 and M_2, M_2 gets to know the bill amount sent by M_1 and this can be asserted using an $\ominus_{M_1}$ formula in M_2. Accordingly M_2 can change its bill and send it to S. The Lamport diagram fragment in the Figure 2.3 explains the scenario. The formula is as follows:

$$(\ominus_B snd\_quote\_req \wedge \ominus_{M_1} snd\_B3\_1) \supset \bigcirc(snd\_B1\_i \oplus snd\_B2\_i)$$

which means, if the buyer has asked for a quote and M_1 is sending bill worth $B3$, then i shall send a bill with value either $B1$ or $B2$.

2.5 Satisfiability Problem

wm-LTL, essentially, is a composition of two logics, m-LTL [67] and w-LTL, the logic whose satisfiability and model checking issues we shall explore in the next chapter. The i-local formulas for m-LTL are

$$\Phi_i ::= p \ \in \ P_i \mid \neg\, \alpha \mid \alpha_1 \ \vee \ \alpha_2 \mid \bigcirc\, \alpha \mid \alpha_1 \ \mathbf{U} \ \alpha_2 \mid \ominus_j \beta$$

whereas i-local formulas for w-LTL are

$$\Phi_i ::= p \ \in \ P_i \mid \neg\, \alpha \mid \alpha_1 \ \vee \ \alpha_2 \mid \bigcirc\, \alpha \mid \alpha_1 \ \mathbf{U} \ \alpha_2 \mid \diamondminus\, \alpha \mid \diamondminus_j \beta \mid \langle now \rangle_j \beta.$$

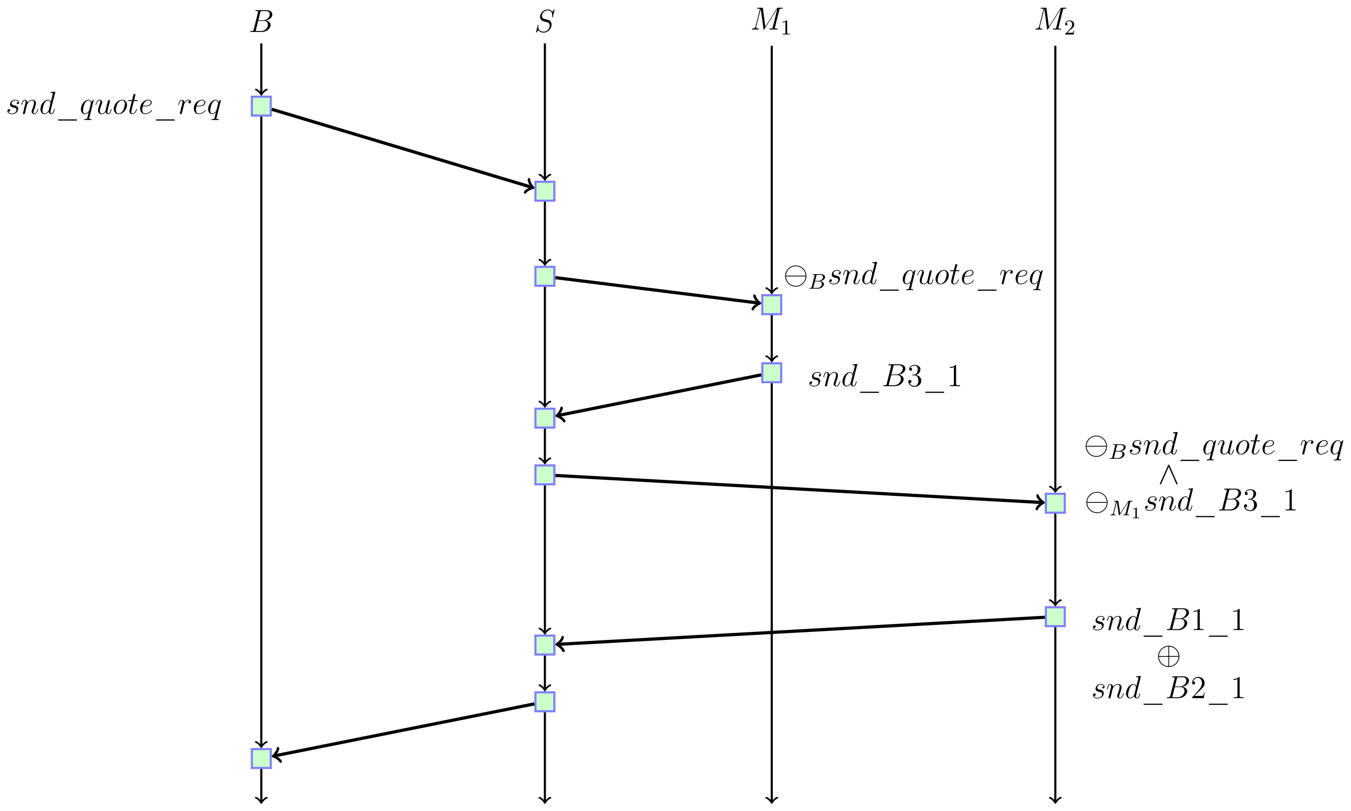


Figure 2.3: Fragment of a quote-request WS execution patterns

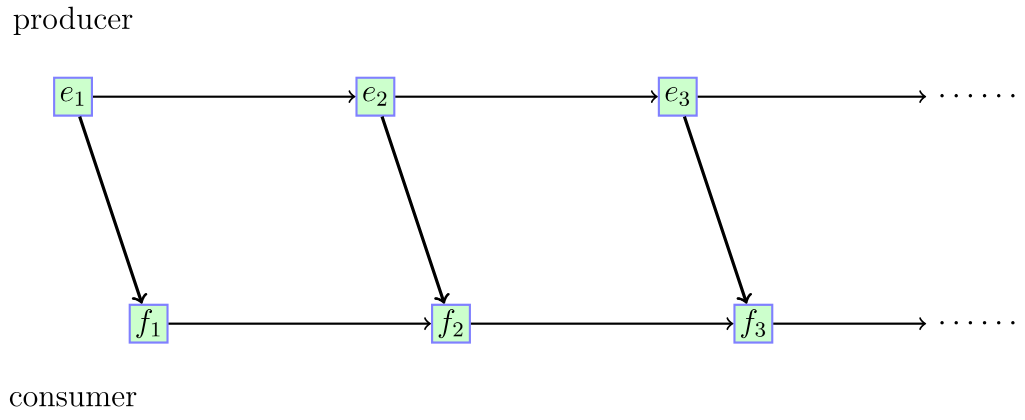


Figure 2.4: Lamport diagram of the producer-consumer protocol

In [66] the satisfiability problem for m-LTL is shown to decidable by the standard Vardi - Wolper technique of constructing an automaton A_ψ for every formula ψ such that the models of ψ correspond to the language accepted by A_ψ. If checking non-emptiness is decidable for this class of automata, we get a decision procedure. We shall use the same technique to tackle the decidability issues of w-LTL. In both the cases, the decidability argument crucially depends on the fact that every labelled Lamport diagram which is a model of a w-LTL formula (or an m-LTL formula) has a 1-bounded sequentialization.

2.5.1 Sequentializations of Lamport diagrams and wm-LTL

Given a Lamport diagram $D = (E, \leq, V)$, a sequentialization of D is any sequence $\sigma = e_0 e_1 \ldots$ such that $E = \{e_0, e_1, \ldots\}$ and for all $k \geq 0$, $\downarrow e_k \subseteq \{e_0, \ldots, e_k\}$; that is, σ is a linear order that respects $\leq$.

For example, consider the Lamport diagram D corresponding to the producer consumer protocol given in the Figure 2.4. The events $e_1, e_2, e_3 \cdots$ are those of producer and $f_1, f_2, f_3 \cdots$ are those of consumer. For all $k \geq 1$, e_k is a send event from producer to consumer, whereas, f_k is the corresponding receive event. As an example the sequence $\sigma_1 = e_1 f_1 e_2 f_2 \cdots$ is a valid sequentialization of D , whereas, the sequence $\sigma_2 = f_1 e_1 e_2 f_2 \cdots$ is not a legal sequentialization as $\downarrow f_1 \not\subseteq \{f_1\}$.

It is not difficult to observe that a Lamport diagram can be represented by the set of all its sequentializations. This, consequently, defines a language of sequences of events (sequentializations) $Lin(D)$ of the Lamport Diagram D. Similarly, a collection of Lamport diagrams can be represented by the set of all sequentializations of each Lamport diagram in the collection.

Proposition 2.5.1 ([66]). *A Lamport diagram can be represented by the set of all its sequentializations.*

Proof. Consider the set $Lin(D)$ of all sequentializations of some Lamport diagram $D = (E, \leq, V)$. Every sequentialization in the set $Lin(D)$ is an infinite string over E, so in order to show that D can be fully represented by $Lin(D)$, it suffices to show that the causal order $\leq$ can be recovered from the set $Lin(D)$.

Fix a sequentialization $\sigma \in Lin(D)$ and consider two events e_1 and e_2 in σ such that e_1 occurs before e_2 in the sequentialization. If e_1 occurs before e_2 in every other sequentialization in $Lin(D)$ then it is easy to see that $e_1 \leq e_2$ in D. Suppose not, *i.e.*, either $e_2 \leq e_1$ or e_1 and e_2 are unordered in D. In the former case, it contradicts the fact that σ *is* a sequentialization of D. In the latter case, there would be some sequentialization σ' where e_2 may come before e_1. This violates the assumption that e_1 precedes e_2 in *all* sequentializations of D.

Consider e_1 and e_2 in σ again. Suppose there is another sequentialization in $Lin(D)$, say σ', where e_2 precedes e_1. In this case, e_1 and e_2 are unordered in D. Suppose not. That is, either $e_1 \leq e_2$ or $e_2 \leq e_1$. In the former case, *all* sequentializations will see e_1 preceding e_2, but σ' violates it, whereas, in the latter case, *all* sequentializations will see e_2 preceding e_1, but σ violates it.

Thus, we can define $\leq$ by looking at $Lin(D)$. Clearly, $Lin(D)$ is an alternate representation of D. □

Sequentializations of a Lamport diagram induce the notion of a *buffer* between agents of the system. The buffer records the sequence of “pending sends” between every pair of agents for every prefix of the given sequentialization. For example, the sequentialization σ_1 of D given above is 1-*bounded* as there is at most one send event e_k without the corresponding receive event f_k in every prefix of σ_1. The sequentialization $\sigma_3 = e_1 e_2 f_1 f_2 \ldots$ is at least 2-*bounded* as the prefix $e_1 e_2$ of σ_3 has two send events e_1 and e_2 without their corresponding receive events f_1 and f_2. We interpret e_1 and e_2 as *pending send events* at the prefix $e_1 e_2$ of σ_3. A sequentialization where the number of pending send events grows unbounded as we consider prefixes of increasing length is *unbounded*. We are interested in sequentializations which implicitly use a "bounded buffer" with the hope that these will be "regular" languages. Let $\sigma = e_0 e_1 \ldots$ be a sequentialization of D. We say σ is b-bounded (for $b \in \mathbb{N}$) iff the following property holds: Consider events $e_1, e_2, \ldots e_{b+1}$ and $f_1, f_2, \ldots f_{b+1}$, $e_k \in E_i$, $1 \leq_{\mathbb{N}} k \leq_{\mathbb{N}} (b+1)$ and $f_k \in E_j$, $1 \leq_{\mathbb{N}} k \leq_{\mathbb{N}} (b+1)$ where $i, j \in [n]$ with $i \neq j$ such that:

1. $e_1 \leq_i e_2 \leq_i \cdots \leq_i e_{b+1}$,
2. $f_1 \leq_j f_2 \leq_j \cdots \leq_j f_{b+1}$,
3. $e_k <_c f_k$ for $1 \leq_{\mathbb{N}} k \leq_{\mathbb{N}} (b+1)$ and
4. for every k, $1 \leq_{\mathbb{N}} k \leq_{\mathbb{N}} b$, if there exists no event $e \in E_i$ and no event $f \in E_j$ such that $e_k <_i e <_i e_{k+1}$, $f_k <_j f <_j f_{k+1}$ and $e <_c f$ then f_b comes before e_{b+1} in σ.

That is, if $e_1, \cdots e_{b+1} \in E_i$ and $f_1, \cdots, f_{b+1} \in E_j$ are $b+1$ consecutive matching send-receive pairs between agent i and j then f_b comes before e_{b+1} in a b-bounded σ. That is, a b-bounded sequentialization is one in which for every pair of distinct agents i, j, at most b send events from i to j can occur in any prefix of the sequentialization without their corresponding receive events having occurred. Lamport diagrams, in general, may not have 1-bounded sequentializations. For example, the 2-agent Lamport diagrams in Figure 2.5 do not have any. The Lamport diagrams in Figure 2.5 can be transformed to 4-agent Lamport diagrams with 1-bounded sequentializations, as in Figure 2.6. The extra agents work as buffers between the two agents and make sure that there never is more than one pending send event in either of the two original agents. In general, an n-agent Lamport diagram can be transformed into an $(n + n(n-1))$-agent Lamport Diagram admitting only 1-bounded sequentializations.

Suppose the agents in the original Lamport diagram are labelled from the set $[n] = \{1, \cdots, n\}$. Then, the transformed Lamport diagram has additional agents labelled (j, j'), for all $j \neq j' \in [n]$, apart from the original ones. Notice that, in such a scenario, a send-to-j' event in agent j in the original diagram becomes a synchronization between agent j and agent (j, j') in the transformed diagram. Similarly, a receive-from-j' event in agent j becomes a synchronization between

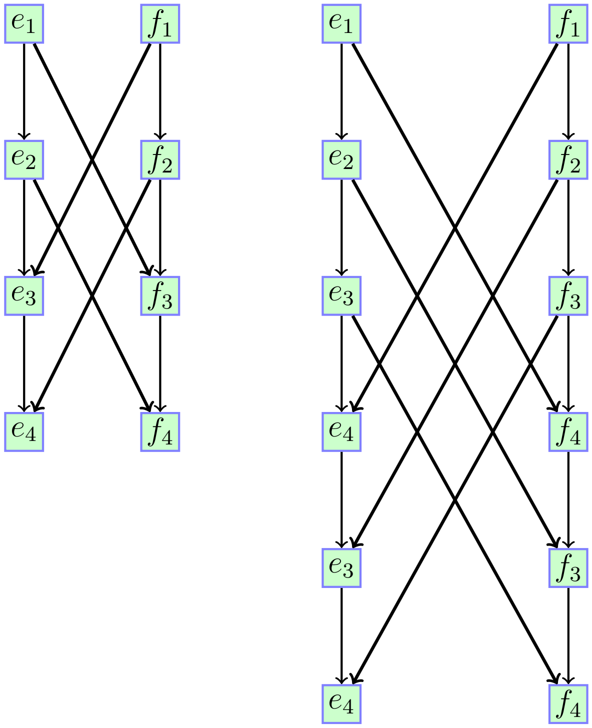


Figure 2.5: Lamport diagrams with no 1-bounded sequentializations

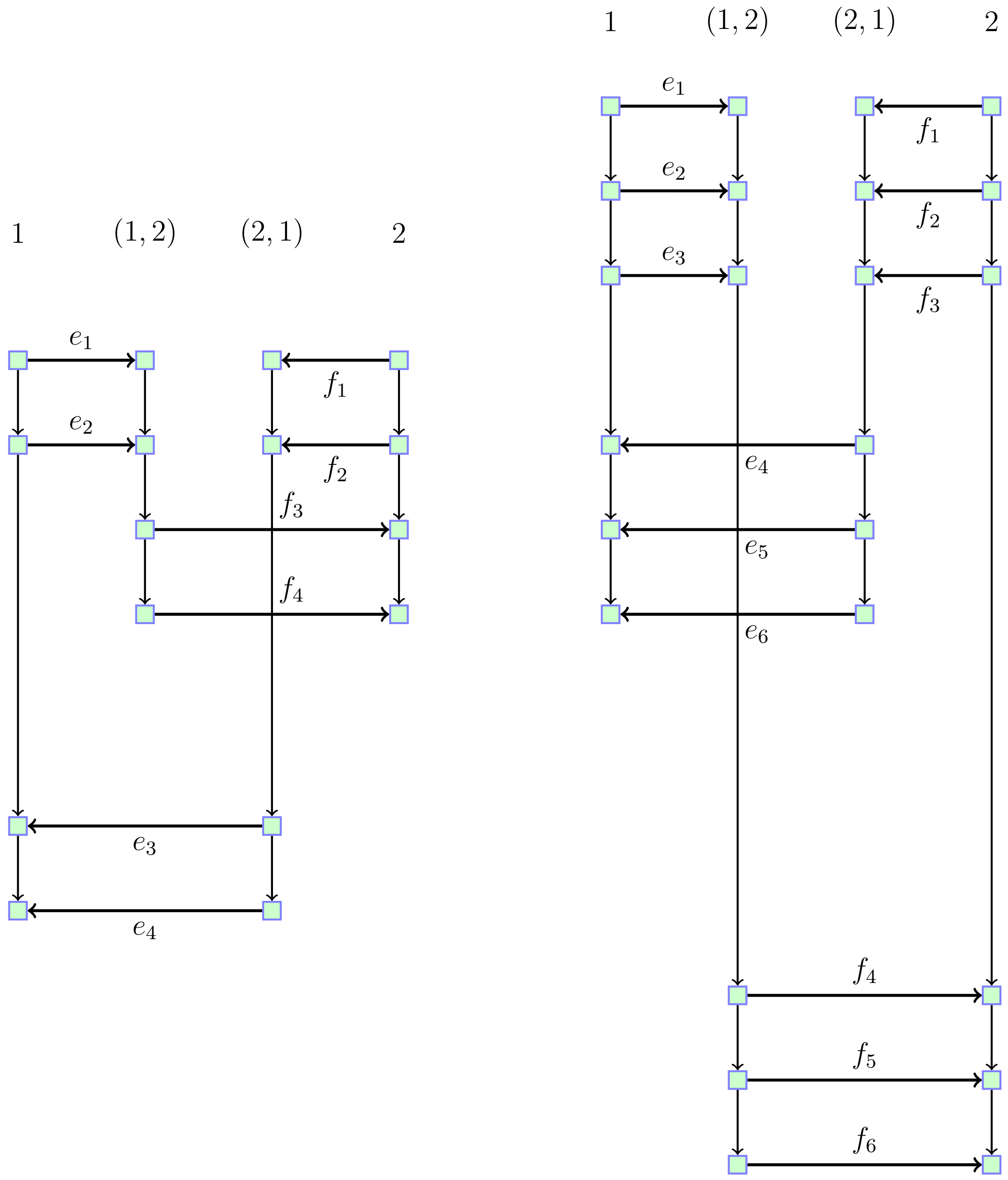


Figure 2.6: Transformed Lamport diagrams with 1-bounded sequentializations

agent j and agent (j',j). The local events in j remain as they are. Given a Lamport diagram $D=(E,\leq,V)$ over distributed alphabet $\widetilde{\Sigma}=(\Sigma_1,\cdots,\Sigma_n)$ we can define a transformed diagram TrD over distributed alphabet $\widetilde{Tr\Sigma}=(\widetilde{Tr\Sigma_a},\widetilde{Tr\Sigma_c})$.

- $\widetilde{Tr\Sigma_a}=(Tr\Sigma_1,\cdots,Tr\Sigma_n)$ is the alphabet of agents, where for all $1\leq j\leq n$, $Tr\Sigma_j=\Sigma_j\times\{j\}$ and
- $\widetilde{Tr\Sigma_c}=(Tr\Sigma_{1,2}\cdots,Tr\Sigma_{1,n},\cdots,Tr\Sigma_{n,1}\cdots,Tr\Sigma_{n,(n-1)})$ is the alphabet of channels, where for all $1\leq j\neq j'\leq n$, $Tr\Sigma_{j,j'}=\Sigma_j\cup\Sigma_{j'}\times\{(j,j')\}$

The transformed diagram is another triple $TrD=(E',\leq',V')$ where

- The event set of transformed diagram remains the same as that of Lamport diagram, *i.e.*, $E'=E$
- Also, the causality relation does not change. That is, $\leq'=\leq$.
- Only the labels change. Before we describe the labelling we define ${}^\bullet e$ and $e^\bullet$ for each $e\in E$. Given $e\in E$, ${}^\bullet e=\{e'\in E\mid e'<_c e\}$ is nonempty if e is receive event and $e^\bullet=\{e'\in E\mid e<_c e'\}$ is non-empty when e is a send event. We assume that these two sets, for a given $e\in E$ can be at most singleton. That is, either e is a local event or it is a send-to-j' event for a unique j' or it is a receive-from-j' event for a unique j'.

  Now, we are ready the define the map V', $V':E'\to Tr\Sigma$ such that

$$V'(e)=\begin{cases}(V(e),j) & \text{if } V(e)\in\Sigma_j \text{ and } {}^\bullet e\cup e^\bullet=\emptyset\\ (V(e),\mathbf{s},(j,j')) & \text{if } V(e)\in\Sigma_j \text{ and } e^\bullet\cap E_{j'}\neq\emptyset\\ (V(e),\mathbf{r},(j',j)) & \text{if } V(e)\in\Sigma_{j'} \text{ and } {}^\bullet e\cap E_j\neq\emptyset\end{cases}$$

As in the case of Lamport diagrams, we can define local states corresponding to each $e\in E'$ as the downclosed set $\downarrow e$. Also, we can define local states for each agent in TrD. Furthermore, we can define events for each agent in TrD. That is, E'_j's for each $j\in[n]$ and $E'_{j,j'}$ for each $j\neq j'\in[n]$ as follows:

$$E'_j=\{e\in E'\mid V'(e)[2]=j \text{ or } V'(e)[3]=(j,j') \text{ for some } j'\neq j\in[n]\},$$

$$E'_{j,j'}=\{e\in E'\mid V'(e)[3]=(j,j')\},$$

$$LC'_j = \{\emptyset\} \cup \{{\downarrow}e \mid e \in E'_j\} \text{ and}$$

$$LC'_{j,j'} = \{\emptyset\} \cup \{{\downarrow}e \mid e \in E'_{j,j'}\}.$$

Let us say a few words about the local states of the channel agents in the transformed diagrams. Consider the transformed diagram on the left in Figure 2.6. The set of local states of the channel agent $(1, 2)$ is as follows:
$LC_{1,2} = \{\emptyset, \{e_1\}, \{e_1, e_2\}, \{e_1, e_2, f_3\}, \{e_1, e_2, f_3, f_4\}\}$, where $V'(e_1) = (V(e_1), \mathbf{s}, (1, 2))$, $V'(e_2) = (V(e_2), \mathbf{s}, (1, 2))$, $V'(f_3) = (V(f_3), \mathbf{r}, (2, 1))$ and $V'(f_4) = (V(f_4), \mathbf{r}, (2, 1))$. Observe that for every local state $d' \in LC_{1,2}$, the number of events in d' with labels having a $\mathbf{s}$ are greater than or equal to the number of events with labels having a $\mathbf{r}$. This encodes the requirement that the channel agent $(1, 2)$ in a state d' can synchronize with agent 2 via a receive-from-1 event in 2 only if there is a pending send in the buffer, that is an event $e \in d'$ with a $\mathbf{s}$ in its label and no matching event $f \in d'$ with a $\mathbf{r}$ in its label.

Clearly, TrD projected to $[n]$ is the same as D. That is, for all $j \in [n]$, $LC_j = LC'_j$ and $E'_j = E_j$. Also $\leq'$ over $[n]$ is the same as $\leq$. The changes which are effected are as follows: The send-to-j' events in E_j become common events with the channel (agent) (j, j') and receive-from-j' events in E_j become common events with the channel (j', j). Consequently, only the labelling of events change when we go from D to TrD.
Now it is easy to show that:

Proposition 2.5.2. *Given any Lamport diagram D, the transformed diagram TrD has a 1-bounded sequentialization.*

It turns out that a Lamport diagram D and the corresponding transformed diagrams TrD have identical properties vis-a-vis the logic *wm*-LTL. In other words, the formulas of *wm*-LTL cannot distinguish between a Lamport diagram and its corresponding transformed diagrams. This fact can be formalized by showing that for a given Lamport diagram D, D and TrD are **bisimilar**.

Before we proceed, we define satisfiability of *wm*-LTL formulas over transformed diagrams. Local satisfiability $\models_j$, for each j-local state $d' \in LC'_j$ in TrD, over j-local formulas $\alpha \in \Phi_j$ is defined, as usual, by induction over structure of α with the following base case. The inductive cases remain exactly the same as for

Lamport diagrams.

$$TrD, d' \models_j p \text{ iff } p \in V'(d)[1].$$

Global satisfiability $\models$ over global formulas $\psi \in \Psi$ too is defined, inductively, exactly as we did for Lamport diagrams.

Also, we define a bisimulation relation between Lamport diagrams with different number of agents, namely n-agent and $(n+m)$-agent Lamport diagrams. Suppose D is the n-agent Lamport diagram and D' the $(n+m)$-agent Lamport diagram. The relation $\sim_{[n]}$ relates the properties of local states of all the agents in D with only the local states of n of the agents in D'.

Definition 2.5.3. *Given an n-agent Lamport diagram $D = (E, \leq, V)$ and a $(n+m)$-agent Lamport diagram $D' = (E', \leq', V')$, a non-empty binary relation $\sim_{[n]} \subseteq LC \times LC'$ is a* bisimulation *if and only if for all $j \in [n]$ for all $d \in LC_j, d' \in LC'_j$, $(d \sim_{[n]} d')$ when the conditions given in definition 2.3.1 hold.*

Consequently, bisimilarity between a n-agent Lamport diagram and $n{+}m$-agent Lamport diagram is defined as follows:

Definition 2.5.4. *Given an n-agent Lamport diagram D and a $n+m$-agent Lamport diagram D', they are* bisimilar *if there exists a bisimulation $\sim_{[n]}$ over D and D' such that for every $j \in [n]$ $(\varepsilon_j \sim_{[n]} \varepsilon'_j)$.*

Now, the following lemma about the bisimilar invariance of wm-LTL can be is easily proved using induction on the structure of the formulas:

Lemma 2.5.5. *Given an n-agent Lamport diagram D and a $n+m$-agent Lamport diagram D', if $D \sim_{[n]} D'$ then for all wm-LTL formula ψ $D \models \psi$ iff $D' \models \psi$.*

Given an arbitrary labelled n-agent Lamport diagram $D = (E, \leq, V)$ we construct an $(n+n(n-1))$-agent transformed diagram $TrD = (E', \leq', V')$ which has a 1-bounded sequentialization and is bisimilar with D. Now it is easy to show that:

Lemma 2.5.6. *Given a labelled Lamport diagram $D = (E, \leq, V)$ there is a $TrD = (E, \leq, V')$ with 1-bounded linearization such that $D \sim_{[n]} TrD$.*

Let LC be the set of all local states of D and LC' be the local states of TrD. Then, $\sim_{[n]}$ is defined as follows: for all $d \in LC$, for all $d' \in LC'$, $d \sim_{[n]} d'$ iff there exists $e \in E$ such that $d = \downarrow e$ in D and $d' = \downarrow e$ in TrD.

Thus, we can now assert the following:

Lemma 2.5.7. *For any wm-LTL formula ψ and for any Lamport Diagram D,*

$$D \models \psi \textit{ iff } TrD \models \psi.$$

Theorem 2.5.8. *Let $D = (E, \leq, V)$ be a labelled Lamport diagram which is a model of wm-LTL formula ψ. then we can define a transformed diagram, as described above, $TrD = (E', \leq', V')$ such that D and TrD are **bisimilar**, TrD has a 1-bounded sequentialization and $TrD \models \psi$.*

Now, for any *wm*-LTL formula ψ_0, let *Models*(ψ_0) be the set of all Lamport diagrams which are models of ψ_0. As defined above, let *TrModels*(ψ_0) be the corresponding set of 1-bounded bisimilar transformed diagrams. It is easy to observe the following:

Lemma 2.5.9. *Models*(ψ_0) $\equiv$ *TrModels*(ψ_0).

Note, that, the sets *Models*(ψ_0) and *TrModels*(ψ_0) are not equal, they are equivalent. That is, for every $D \in$ *Models*(ψ_0) we can construct $TrD \in$ *TrModels*(ψ_0) and vice versa.

This motivates us to work with a distributed automaton model, called SCA running on Lamport diagrams, which when capturing policies specified by a logic like *wm*-LTL has a 1-bounded Büchi equivalent consuming the corresponding 1-bounded sequentializations. We now present this class of automata and then proceed to consider the formula automaton construction.

3

Decidability of w-LTL

In this chapter we investigate the satisfiability problem for the logic w-LTL. The language w-LTL is obtained from wm-LTL by dropping the $\ominus_j$ modality. When we remove $\langle now \rangle_j$ modality instead of $\ominus_j$ we get the language m-LTL which was shown to be decidable in [67]. We add special i-local propositions in w-LTL of the form $\mathbf{s}^j$ denoting "send-to-j"-events, $\mathbf{r}^j$ denoting "receive-from-j" and $\mathbf{no\_comm}^j$ denoting "no-further-communication-with-j"-events for the sake of technical convenience in construction of formula automaton for a given formula ψ_0. These propositions take the following meaning:

- $M, d \models_i \mathbf{s}^j$ iff $\exists d' \in LC_j$ such that $d <_c d'$.
- $M, d \models_i \mathbf{r}^j$ iff $\exists d' \in LC_j$ such that $d' <_c d$.
- $M, d \models_i \mathbf{no\_comm}^j$ iff $\forall d' \in LC_i$, $d \leq d'$ $\neg\big(\exists d'' \in LC_j$ such that $d' <_c d''$ or $\exists d'' \in LC_j$ such that $d'' <_c d'$ $\big)$.

The satisfiability problem for w-LTL is settled using Systems of Communicating Automata (SCA), a class of CFSMs [16], and following [84], by effectively associating an SCA S_ψ with each formula ψ in such a way that

$$Lang(Pr_{S^1_\psi}) \neq \emptyset \text{ iff } TrModels(\psi) \neq \emptyset$$

where $Pr_{S^1_\psi}$ is the 1-product automaton of S_ψ obtained by explicitly buffering messages in the channels between the n agents mentioned in the formula ψ. Thus, the satisfiability of ψ can be checked by looking at the emptiness of the product language of S_ψ. Now, as we observe that using w-LTL we can never write a formula

ψ which enforces models with no 1-bounded linearization in *Models*(ψ), checking the emptiness of 1-bounded product of S_ψ, $Pr^1_{S_\psi}$, which is a Büchi automaton [18], would suffice instead. As we know how to check language emptiness of a Büchi automata in an efficient manner [81] the satisfiability of w-LTL turns out to be decidable.

We describe SCAs and their bounded counterparts (1-product *etc.*,) in Büchi automata, in detail, in this chapter. We also give a formula automaton construction for w-LTL and show its correctness.

3.1 System of Communicating Automata

Systems of Communicating Automata (SCA) were introduced in [67] and the presentation here follows [67] and [66].

As before, we fix $n > 0$ and focus our attention on n-agent systems. A distributed alphabet for such systems is an n-tuple $\widetilde{\Sigma} = (\Sigma_1, \ldots, \Sigma_n)$, where for each $i \in [n]$, Σ_i is a finite non-empty alphabet of actions of agent i and for all $i \neq j$, $\Sigma_i \cap \Sigma_j = \emptyset$. The alphabet induced by $\widetilde{\Sigma} = (\Sigma_1, \ldots, \Sigma_n)$ is given by $\Sigma = \bigcup\limits_i \Sigma_i$. The set of system actions is the set $\Sigma' = \{\lambda\} \cup \Sigma$. The action symbol λ is referred to as the communication action. This is used as an action representing a communication constraint through which every receive action will be dependent on its corresponding send action. We use a, b, c *etc.*, to refer to elements of Σ and τ, τ' *etc.*, to refer to those of Σ'.

Definition 3.1.1. *A* **System of** n **Communicating Automata (SCA)** *on a distributed alphabet* $\widetilde{\Sigma} = (\Sigma_1, \ldots, \Sigma_n)$ *is a tuple* $S = ((Q_1, G_1), \ldots, (Q_n, G_n) \rightarrow, Init)$ *where,*

1. *For* $j \in [n]$, Q_j *is a finite set of (local) states of agent* j.
   For $j \neq j'$, $Q_j \cap Q_{j'} = \emptyset$.

2. $Init \subseteq (Q_1 \times \ldots \times Q_n)$ *is the set of (global) initial states of the system.*

3. *for each* $j \in [n]$, $G_j \subseteq Q_j$ *is the set of (local) good states of agent* j.

4. *Let* $Q = \bigcup\limits_j Q_j$, *then, the transition relation* $\rightarrow$ *is defined over* Q *as follows.* $\rightarrow \subseteq (Q \times \Sigma' \times Q)$ *such that if* $q \xrightarrow{\tau} q'$ *then either there exists* j *such that*

$\{q, q'\} \subseteq Q_j$ and $\tau \in \Sigma_j$, or there exist $j \neq j'$ such that $q \in Q_j, q' \in Q_{j'}$ and $\tau = \lambda$.

Thus, SCAs are systems of n finite state automata with λ-labelled communication constraints between them. Note that $\rightarrow$ above is *not* a global transition relation, it consists of local transition relations, one for each agent, and communication constraints of the form $q \xrightarrow{\lambda} q'$, where q and q' are states of different agents. The latter define a coupling relation rather than a transition. The interpretation of local transition relations is standard: when the agent i is in state q_1 and reads input $a \in \Sigma_i$, it can move to a state q_2 and be ready for the next input if $(q_1, a, q_2) \in \rightarrow$. The interpretation of communication constraints is non-standard and depends only on automaton states, not on local input. When $q \xrightarrow{\lambda} q'$, where $q \in Q_i$ and $q' \in Q_j$, it constrains the system behaviour as follows: whenever agent i is in state q, it puts a message whose content is q and intended recipient is j into the buffer; whenever agent j intends to enter state q', it checks its environment to see if a message of the form q from i is available for it, and waits indefinitely otherwise. If a system S has no λ constraints at all, automata proceed asynchronously and do not wait for each other. We will refer to λ-constraints as 'λ-transitions' in the sequel for uniformity, but this explanation (that they are constraints not dependent on local input) should be kept in mind.

We use the notation ${}^{\bullet}q \overset{\text{def}}{=} \{q' \mid q' \xrightarrow{\lambda} q\}$ and $q^{\bullet} \overset{\text{def}}{=} \{q' \mid q \xrightarrow{\lambda} q'\}$. For $q \in Q$, the set ${}^{\bullet}q$ refers to the set of all states from which q has incoming λ-transitions and the set $q^{\bullet}$ is the set of all states to which q has outgoing λ-transitions. *Global behaviour* of an SCA will be defined using its set of global states. To refer to global states, we will use the set $\widetilde{Q} \overset{\text{def}}{=} (Q_1 \times \cdots \times Q_n)$. When $u = (q_1, \ldots, q_n) \in \widetilde{Q}$, we use the notation $u[i]$ to refer to q_i.

Figure 3.1 gives an SCA over the alphabet $\widetilde{\Sigma} = (\{a\}, \{b\})$ (We use $\Rightarrow$ to mark the initial states). The (global) initial states and (global) good states of this SCA are $\{(s_0, t_0)\}$. The reader will observe that this SCA models the producer-consumer protocol. The producer generates an object via a $s_0 \xrightarrow{a} s_0$ transition, whereas the consumer consumes an object through a $t_0 \xrightarrow{b} t_0$ transition. As a consumption can follow only after a production there is a λ transition between s_0 and t_0. Now, it is not difficult to see why (s_0, t_0) is an initial state as well as final state. The producer can always terminate in s_0 after generating zero or more objects and consumer can

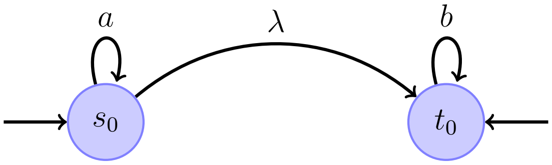


Figure 3.1: A simple SCA

terminate in t_0 after consuming one or more objects.

The language accepted by an SCA is a collection of (Σ-labelled) Lamport diagrams.

3.1.1 Poset language of an SCA

We now formally define run of an SCA on a Lamport diagram and the poset language accepted by an SCA as the collection of Lamport diagrams on which the SCA has an accepting run.

Given an SCA S on $\widetilde{\Sigma}$, a run of S on a Lamport diagram $D = (E, \leq, V)$ is a map $\rho : C_D^{fin} \to \widetilde{Q}$ such that the following conditions are satisfied:

- $\rho(\emptyset) \in Init$.
- For $c \in C_D^{fin}$, suppose $\rho(c) = (q_1, q_2, \ldots, q_n)$. Consider $c' = (c \cup \{e\}) \in C_D^{fin}$, where $e \in E_i, e \notin c$ such that $V(e) = a \in \Sigma_i$. Then,
  - $\rho(c') = (q'_1, q'_2, \ldots, q'_n)$ where $q'_j = q_j$ for all $j \neq i$ and $q_i \xrightarrow{a} q'_i$ in S.
  - Suppose $\exists e' \in E_j$, $j \neq i$ such that $e' \lessdot e$ then there exists $q \in Q_j$ and $c'' \subseteq \downarrow e'$ such that $\rho(c'')[j] = q$, $q \xrightarrow{b} \rho(\downarrow e')[j]$ and $q \xrightarrow{\lambda} q'_i$.
  - If ${q_i}^{\bullet} \cap Q_j \neq \emptyset$, then, there exists $e' \in E_j$ such that $e \lessdot e'$.

Thus, a run of S on D is a map from the set C_D^{fin} of configurations of D to the set of global states of S such that the following conditions hold: If c' is a configuration obtained by adding an event $e \in E_i$ (where $V(e) = a$) to a configuration c then, there is a transition on a from the local state of agent i in $\rho(c)$ to the local state of the same agent in $\rho(c')$ and all other local states are unaltered. In addition, if e is a receive event, we ensure that the corresponding send event has already occurred

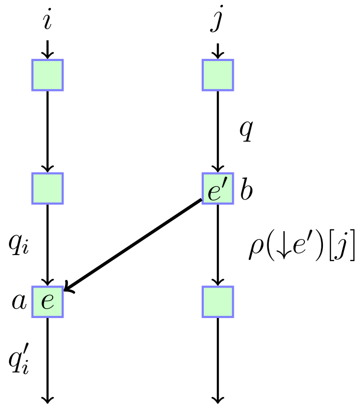


Figure 3.2: Updates in $\rho(c')$ from $\rho(c)$

and that there is a λ-constraint into the resulting state. Note, that, if there are out-going λ-constraints from the enabling state, the definition makes sure that the corresponding event e is a send event and that it has a matching receive event. In order to define goodness of a run, we first define terminal states. A state $q \in Q_i$ is terminal in i if $\{q' \mid q \xrightarrow{a} q' \text{ for some } a \in \Sigma_i\} = \emptyset$.

Given a run $\rho : C_D^{fin} \to \widetilde{Q}$ of SCA S on Lamport diagram D, ρ is said to be good if $\forall i \in [n]$, E_i is finite implies $\rho(\downarrow e)[i]$ is a terminal state where e is the maximum event (with respect to $\leq$) in E_i. Also, for each $i \in [n]$, we define $Inf_i(\rho)$, the set of all i-local states which occur infinitely many times in the run ρ, as follows: $Inf_i(\rho) = \{q \in Q_i \mid \text{ there exists infinitely many } c \in C_D^{fin} \text{ such that } \rho(c)[i] = q\}$.

Now, we spell out the acceptance condition for a run $\rho : C_D^{fin} \to \widetilde{Q}$ of S over D . The run ρ is said to be accepting if following criteria hold:

- ρ is good and
- for all $i \in [n]$, $Inf_i(\rho) \cap G_i \neq \emptyset$.

The poset language accepted by S is denoted by $\mathcal{L}^{po}(S)$ and is defined as:

$\mathcal{L}^{po}(S) \overset{\text{def}}{=} \{D \mid D \text{ is a Lamport diagram and } S \text{ has an accepting run on } D\}$.

For example, Figure 3.3 gives a run of the SCA in Figure 3.1 over the Lamport diagram of producer-consumer problem given in the Figure 2.4. The figure essentially gives the Directed Acyclic Graph corresponding to the configuration space

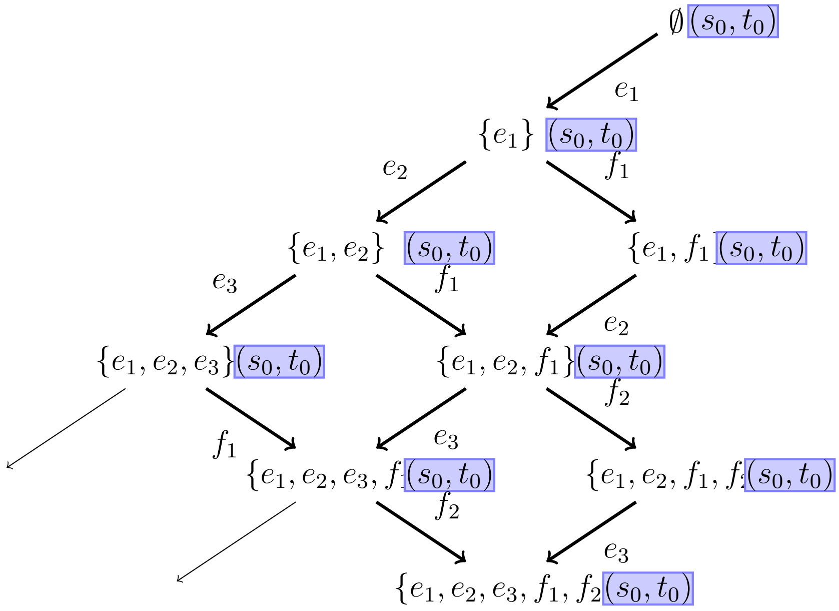


Figure 3.3: The run of SCA over a Lamport diagram

of the Lamport diagram. Each node (configuration) has an associated state label given in shaded boxes on the right.

We note that the class of poset languages accepted by SCAs is easily seen to be closed under union and intersection, since the automata are nondeterministic.

Proposition 3.1.2 ([66])**.** *SCAs are closed under union and intersection.*

As mentioned before, we will be using SCAs to show decidability of the satisfiability problem for w-LTL. Towards this, we now address the problem of checking if the poset language accepted by a given SCA is non-empty and show that it is decidable. A standard approach to solve the emptiness problem for sequential finite state automata is to look for strongly connected components containing a good state (in the graph of the automaton) which are reachable from one of the initial states. Towards using this approach for SCAs, we define the global automaton corresponding to an SCA by taking products of local states and including the states of buffers. But, the global automaton need not be finite-state in general as buffers can be unbounded. We then note that bounded buffers suffice, using the fact that all labelled Lamport diagrams which are models of w-LTL formulas have 1-bounded sequentializations. Therefore, we can show that buffers of size one suffice to show language emptiness of an SCA.

3.1.2 $*$-Products of SCAs

The global automaton corresponding to a given SCA is defined by taking the products of local automata (representing parallel composition of sequential behaviours) and storing pending messages in buffers. We also make sure that actions corresponding to send events have appropriate actions which represent their matching receive events. The buffers are represented as queues of arbitrary length and are meant to store *pending messages* between agents. There is a transition from one global state to another on an action of agent i if and only if there is a corresponding (local) transition on *that* action in the automaton of agent i. In addition, the buffers are updated depending on whether the action represents a send or a receive, that is, whether there is a λ-transition going out or coming in for *that* transition.

We first define buffers and buffer operations **insert** and **delete**.

Definition 3.1.3. *Given an n-SCA $S = ((Q_1, G_1), \dots, (Q_n, G_n) \rightarrow, Init)$, let $\mathcal{Q} = (\bigcup\limits_{j} \mathcal{Q}_j) \cup \{\bot\}$, where $\mathcal{Q}_j = \bigcup\limits_{0 \leq l} Q_j^l$ and $Q_j^l = Q_j \times Q_j \times \cdots$ l times. A buffer Bf is a map $Bf : [n] \times [n] \rightarrow \mathcal{Q}$ such that the following conditions hold:*

1. *for all $j \in [n]$, $Bf(j, j) = \bot$ and*
2. *for all $j \neq j' \in [n]$, $Bf(j, j') \in \mathcal{Q}_j$.*

*A buffer Bf is in **initial** state if for all $j \neq j' \in [n]$, $Bf(j, j') = \epsilon$. Let us denote this unique map by Bf_ϵ. Let $\mathcal{B}$ be the set of all buffers.*

We define two operations **insert** *and* **delete** *on buffers as follows: Given two buffers $Bf, Bf' \in \mathcal{B}$, $j \neq j' \in [n]$ and state $q \in Q_j$,* **insert** *$Bf, (j, j', q) = Bf'$ where*

1. *$Bf'(j, j') = Bf(j, j') \cdot q$,*
2. *for all $j_0 \neq j \neq j' \in [n]$, $Bf'(j, j_0) = Bf(j, j_0)$ and $Bf'(j_0, j') = Bf(j_0, j')$ and*
3. *for all $j_0 \neq j_0' \neq j \neq j' \in [n]$, $Bf'(j_0, j_0') = Bf(j_0, j_0')$.*

delete *$Bf, (j, j', q) = Bf'$ where*

1. *$Bf(j, j') = q \cdot Bf'(j, j')$,*

2. *for all $j_0 \neq j \neq j' \in [n]$, $Bf'(j, j_0) = Bf(j, j_0)$ and $Bf'(j_0, j') = Bf(j_0, j')$ and*

3. *for all $j_0 \neq j'_0 \neq j \neq j' \in [n]$, $Bf'(j_0, j'_0) = Bf(j_0, j'_0)$.*

Definition 3.1.4. *Given an SCA $S = ((Q_1, G_1), \dots, (Q_n, G_n) \rightarrow, Init)$, the* **$*$-product** *of the system is defined to be $Pr^*_S = (X, \widetilde{I}, \widehat{G}, \Rightarrow)$ where*

1. *$X = \widetilde{Q} \times \mathcal{B}$.*

2. *$\widetilde{I} = \{(\widetilde{q}, B_\epsilon) \mid \widetilde{q} \in Init\}$ is the set of initial states, and*

3. *$\widehat{G} = (G_1, \cdots, G_n)$*

4. *the transition relation $\Rightarrow \subseteq (X \times \Sigma \times X)$ is defined by: $(q_1, \dots, q_n, Bf) \overset{a}{\Rightarrow} (q'_1, \dots, q'_n, Bf')$, $a \in \Sigma_i$, iff*

   (a) *$q_i \xrightarrow{a} q'_i$, and for all $j \neq i, q_j = q'_j$.*

   (b) *If $({}^{\bullet}q'_i \cap Q_j) = R \neq \emptyset$, then there exists $q \in R$ and $qw \in Q^*_j$ such that $Bf(i, j) = qw$ and $Bf' =$* **delete** *$Bf, (i, j, q)$.*

   (c) *If $(q_i{}^{\bullet} \cap Q_j) \neq \emptyset$ and for $Bf(j, i) = w$, then $Bf' =$* **insert** *$Bf, (j, i, q)$.*

$\mathcal{B}$ is the set of buffers of the system. There is a buffer between every distinct pair of agents i, j and hence there are totally $n(n-1)$ buffers in the system. The contents of the buffer corresponding to the pair (i, j) represents the sequences of local states of agent i which are messages to agent j. Since messages are assumed to be buffered in the FIFO order, we use sequences of local states (with the assumption that the leftmost element represents the top of the buffer) to represent buffers. Condition (b) ensures that whenever a local i-move is dependent on a message from agent j through a λ constraint, the particular state is there at the head of the buffer between i and j and it is utilized by the i-move.

A state $q \in Q_i$ is terminal if $\{q' \in Q_i \mid q \xrightarrow{a} q'$ for some $a \in \Sigma_i\} = \emptyset$. Let $w = a_1 a_2 \dots \in \Sigma^\omega$. We use the notation $w\lceil i$ to denote the restriction of w to Σ_i. Computations of S can also be defined by runs of Pr^*_S on $w \in \Sigma^\omega$. An infinite run $\rho = x_0 x_1 \ldots$ on w is a sequence where for $k \geq 0$, $x_k \overset{a_{k+1}}{\Rightarrow} x_{k+1}$. For a state $x_k = (\widetilde{q}, B) \in X$, we use the notation $x_k[st]$ to refer to $\widetilde{q}$ and $x_k[buf]$ to refer to B.

We say that i terminates in ρ if there exists a k such that $x_k[st][i]$ is terminal. ρ is said to be good if for all $i \in [n]$, $w\lceil i$ is infinite or i terminates in ρ. Let $Inf_i(\rho) \overset{\text{def}}{=} \{\widetilde{q} \in \widetilde{Q} \mid \exists^\infty k \geq 0, x_k[st] = \widetilde{q},\ x_k[buf](i,j) = \epsilon \text{ for all } j \neq i\}$. The run ρ on w is said to be accepting iff ρ is good, $x_0 \in \widetilde{I}$, and for all $i \in [n]$, $Inf_i(\rho) \cap G_i \neq \emptyset$. The string language accepted by Pr^*_S, denoted $\mathcal{L}^*(S) \overset{\text{def}}{=} \{w \in \Sigma^\omega \mid Pr^*_S$ has an accepting run on $w\}$.

Note, that, the global automaton Pr^*_S, for a given S has an infinite state space, because the buffer sizes are unbounded. The state space can be bounded by putting a bound on the size of buffer. Thus, for a given $m > 0$, we can define m-products of S, by modifying the definition of $*$-product, as follows:

Definition 3.1.5. *Given an SCA $S = ((Q_1, G_1), \ldots, (Q_n, G_n), \rightarrow, Init)$, the* m-**product** *of the system is defined to be $Pr^*_S = (X, \widetilde{I}, \widehat{G}, \Rightarrow)$ where*

1. $X = \widetilde{Q} \times \mathcal{B}$.

2. $\widetilde{I} = \{(\widetilde{q}, B_\epsilon) \mid \widetilde{q} \in Init\}$ *is the set of initial states, and*

3. $\widehat{G} = (G_1, \cdots, G_n)$

4. *the transition relation* $\Rightarrow \subseteq (X \times \Sigma \times X)$ *is defined by:* $(q_1, \ldots, q_n, Bf) \overset{a}{\Rightarrow} (q'_1, \ldots, q'_n, Bf')$, $a \in \Sigma_i$, *iff*

   (a) $q_i \xrightarrow{a} q'_i$, *and for all* $j \neq i, q_j = q'_j$.

   (b) *If* $({}^\bullet q'_i \cap Q_j) = R \neq \emptyset$, *then there exists* $q \in R$ *and* $qw \in Q_j^*$, $|w| < m$ *such that* $Bf(i,j) = qw$ *and* $Bf' = \textbf{delete}\ Bf, (i,j,q)$.

   (c) *If* $(q_i{}^\bullet \cap Q_j) \neq \emptyset$ *and for* $Bf(j,i) = w$, $|w| < m$ *then* $Bf' = \textbf{insert}\ Bf, (j,i,q)$.

Here, condition (c) makes sure that agent i records its message for agent j, if any, provided the corresponding buffer is not full.

The computations and language of Pr^m_S, $\mathcal{L}^m(S)$ can be defined exactly as that for $*$-product.

As an example, Figure 3.4 gives the 1-product and 2-product of the producer-consumer SCA given in the Figure 3.1. The set of initial state as well as good

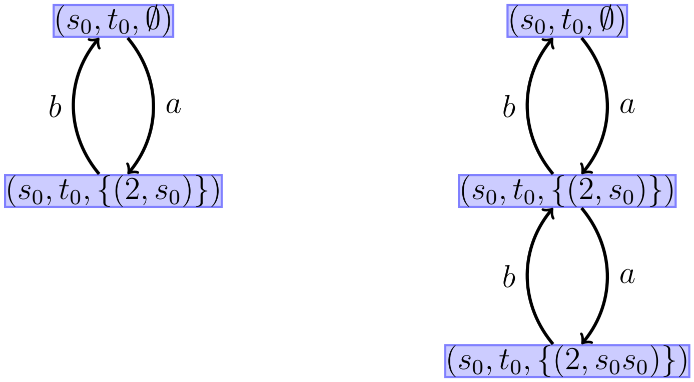

$(s_0, t_0, \emptyset)$
b
a
$(s_0, t_0, \{(2, s_0)\})$
$(s_0, t_0, \emptyset)$
b
a
$(s_0, t_0, \{(2, s_0)\})$
b
a
$(s_0, t_0, \{(2, s_0 s_0)\})$

Figure 3.4: 1-product and 2-product of producer-consumer SCA

states in both the cases is $\{(s_0, t_0, \emptyset)\}$. The language accepted by the 1-product is ab^ω whereas the language accepted by the 2-product is $(ab + aabb + abab)^\omega$.

Language Emptiness of m-product Automata

In this section, we show that the problem of checking if the poset language of a given SCA is non-empty is decidable. As mentioned earlier, given an SCA, we first construct its 1-product and using the fact that Lamport diagrams in the language of the SCA have 1-bounded sequentializations , we show that the poset language of the SCA is non-empty if and only if the language accepted by the corresponding 1-product of the SCA is non-empty. Since the 1-product automaton is a Büchi automaton whose language non-emptiness is decidable, we get decidability of the non-emptiness of the poset language accepted by the SCA.

We first show that the language emptiness of m-product of a given SCA is decidable. Since the m-product is basically a Büchi automaton, language emptiness can be decided by looking for strongly connected components in the underlying graph which contains states from $\widehat{G}$ and which are reachable from the initial states $\widetilde{I}$. This can be done in time linear in the size of product automaton. We thus have:

Lemma 3.1.6. *Given an SCA S of n automata, checking whether $\mathcal{L}^m(S) \overset{?}{=} \emptyset$ can be done in time $k^{O(mn)}$, where k is the maximum of $\{|Q_i| \mid i \in [n]\}$.*

Proof. Given an SCA S, consider its m-product $Pr^m_S = (X, \widetilde{I}, \widehat{G}, \Rightarrow)$. With S, we associate the directed graph $G_S = (V, E)$ with $V = X$ as the set of vertices and $E = \{(x, x') \mid \exists a \in \Sigma,\ x \overset{a}{\Rightarrow} x'\}$ as the set of edges.

A *good component* of G_S is a subset of vertices $V' \subseteq V$ which satisfies the following conditions:

1. There exists $q_0 \in \widetilde{I}$ and there exists $x \in V'$ such that x is reachable from q_0.
2. V' is a maximal strongly connected component.
3. V' satisfies the following condition: there exists an $x \in V'$ such that $x \in \widehat{G}$.

It is easy to check that $\mathcal{L}^m(S) \neq \emptyset$ iff G_S contains at least one good component. The maximal strongly connected components of G_S can be found in time $O(|V|^2)$. If we prove that $|V| = k^{O(mn)}$, we are done.

$|V|$ is the number of states in the m-buffered product which in turn is the product of the number of global states and the number of buffer states. We first estimate the number of buffer states. There are $n(n-1)$ buffers in the system, one for each pair (i, j), $i \neq j$, each containing at most m messages. Therefore, the buffers can be seen as a $n \times n$ matrix with diagonal entries $\perp$ and for all $i \neq j \in [n]$, (i, j)th entry as x_j, a word over Q_j^* of length at most m. Let $|Q_i| = k_i$. Then the number of buffer states $|\mathcal{B}|$ is $\Pi_i(1 + k_i + k_i^2 + \cdots + k_i^m)^{(n-1)} \leq n(1 + k + k^2 + \cdots + k^m)^{n-1}$, where k is the maximum of the k_i's. Therefore the total number of states $|X|$ is at most $(\Pi_i k_i) \cdot (n(1 + k + k^2 + \cdots + k^m)^{n-1}) \leq k^n \cdot n \cdot k^{m(n-1)} = k^{O(mn)}$. □

3.1.3 Emptiness of Poset Language accepted by an SCA

We now establish a $1-1$ correspondence between runs of the 1-product of an SCA and Lamport diagrams in its poset language. Note that this will yield the decidability of emptiness of the poset language of an SCA.

From runs to Lamport diagrams

In this section, we show how to extract Lamport diagrams from computations of 1-products of SCAs. Consider an SCA S over $\widetilde{\Sigma}$ such that its 1-product Pr_S^1 has an (infinite) accepting run $\rho = x_0 x_1 \ldots$, on $w = a_1 a_2 \ldots \in \Sigma^\omega$, *i. e.*, for $k \geq 0$, $x_k \overset{a_{k+1}}{\Rightarrow} x_{k+1}$ in Pr_S^1. We show how to associate a Lamport diagram D_ρ with ρ. We use ρ to define a clock function $\Upsilon : ([n] \times [n] \times \mathbb{N}) \to \mathbb{N}$ which records, for each pair of agents i, j and each instance k, the *latest* instant at which agent i last heard from the agent j at k. Define $\Upsilon(i, j, k)$ by induction on k as follows:

1. For all $i \in [n]$, for all $k \in \mathbb{N}$: $\Upsilon(i, i, k) = k$; for all $i, j \in [n]$, $\Upsilon(i, j, 0) = 0$.
2. Let $k \geq 0$. Suppose $\Upsilon(i, j, k)$ is defined. Let $x_k \overset{a_{k+1}}{\Rightarrow} x_{k+1}$, $a_{k+1} \in \Sigma_i$. Let $j \neq i$. For all $j' \in [n]$, $\Upsilon(j, j', k+1) = \Upsilon(j, j', k)$. Let $\Upsilon(i, j, k) = m$. If ${}^\bullet x_{k+1}[st](i) \cap Q_j = \emptyset$, then $\Upsilon(i, j, k+1) = m$. Otherwise, $x_k = (q_1, \ldots, q_n, Bf)$ and there exists $q \in {}^\bullet x_{k+1}[st](i) \cap Q_j$ such that $Bf(i, j) = q$.

   Claim 3.1.7. *There exists a unique l such that $m < l \leq k$ and $x_l[st](j) = q$.*

   Set $\Upsilon(i, j, k+1) = l$.

The claim is proved as follows. Let $(k, k+1)$ denote the r^{th} receive transition for i from j in ρ. Let $(k'-1, k')$ similarly denote the $(r-1)^{th}$ such transition; if $r = 1$, set $k' = 0$. In either case, by product construction, we have $x_{k'}[buf](j, i) = \epsilon$ and $x_k[buf](j, i) = q$. Let l be the least index such that $k' < l \leq k$ and $x_l[buf](j, i) \neq \perp$. Again by product construction, $x_{l-1}[st](j) = x_l[buf](j, i) = q' \in Q_j$, say. Now let $l \leq l' < k$; we can argue by induction on $l' - l$ that $x_{l'+1}[buf](j, i) = x_{l'}[buf](j, i)$: since no send is enabled when $x_{l'}[buf](j, i) \neq \epsilon$, by the product construction, and there is no receive by choice of indices and these are the only transitions that modify this component. We thus have a unique l such that $x_l[st](j) = q$, as required.

The following proposition follows from our choice of l for $\Upsilon(i, j, k+1)$.

Proposition 3.1.8. *For all $k \geq 0$, for $i \neq j$, $\Upsilon(i, j, k) \leq k$.*

From (ρ, Υ) we can extract a Lamport diagram as follows. Recall that $\rho = x_0 x_1 \ldots$ and for all k, $x_k \overset{a_{k+1}}{\Rightarrow} x_{k+1}$, $a_{k+1} \in \Sigma$. The Lamport diagram is given by $D_\rho \overset{\text{def}}{=} (E, \preceq, V)$, where

1. $E = \{(k, k+1) \mid k \in \mathbb{N}\}$.
2. $V : E \to \Sigma$ is given by $V(e) = a_{k+1}$ iff $e = (k, k+1)$ and $x_k \overset{a_{k+1}}{\Rightarrow} x_{k+1}$ in ρ.
3. $\preceq = (\preceq_l \cup \lessdot_c)^*$, where

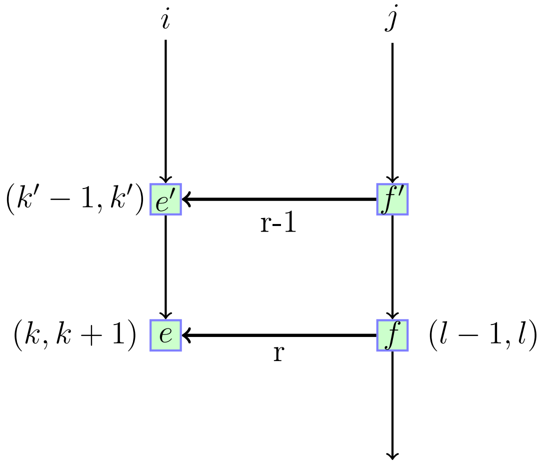


Figure 3.5: Finding unique l for clock function Υ in the run ρ

(a) Let $E_i = \{e \in E \mid V(e) \in \Sigma_i\}$. Then, $\preceq_l = \bigcup_i ((E_i \times E_i) \cap \{((k,k+1),(l,l+1)) \mid k \leq l\})$.

(b) $\lessdot_c = \{((m-1,m),(k,k+1)) \mid$ where $(m-1,m) \in E_j$, $(k,k+1) \in E_i$, $i \neq j$ and $\Upsilon(i,j,k) < \Upsilon(i,j,k+1) = m\}$.

It is easily seen that E_i is linearly ordered by $\preceq$ and that for all e, $\downarrow e \cap E_i$ is finite. Hence, for all e, $\downarrow e$ is finite as well. It only remains to show antisymmetry of $\preceq$. For this, first note that $\lessdot_c$ is asymmetric: whenever $(m-1,m) \lessdot_c (k,k+1)$, by the proposition above, $m < k+1$. Hence $\preceq$ cannot have any cycle that contains a $\lessdot_c$ edge; such a cycle must be composed of i-edges, for some i, violating the fact that E_i is linearly ordered by $\preceq$.

Thus, with each infinite run ρ of Pr^1_S, we can associate a Lamport diagram D_ρ. Hence, by Lemma 3.1.6, we have,

Theorem 3.1.9. *Given an SCA S of n automata, checking whether $\mathcal{L}^{po}(S) \neq \emptyset$ can be done in time $k^{O(n)}$, where k is the maximum of $\{|Q_i| \mid i \in [n]\}$.*

3.2 Satisfiability and Model Checking for w-LTL

The goal of this section is to formulate the model checking problem for w-LTL and show that it is decidable. We again solve this problem using the so-called automata-theoretic approach to model checking. In such a setting, the web service is modelled as an SCA S the specification is given by a formula ψ in w-LTL. The model checking problem is to check if the system satisfies the specification i.e, to check if every "behaviour" of S "satisfies" ψ. To do this, we construct the system S_ψ accepting the models of ψ and check if the poset language of S is a subset of the class of models of ψ. But, the system S, in general is given over some arbitrary alphabet $\widetilde{\Sigma}$ and the system associated with ψ runs over 2^P. So, we have to "interpret" the system S as running over 2^P.

To make this precise, we define an *interpreted system* to be a pair $\mathcal{S} = (S, Val)$, where $S = ((Q_1 \ldots, Q_n), \rightarrow, Init, G)$ on $\widetilde{\Sigma}$, $Val : Q \rightarrow 2^P$ such that for all $q \in Q_i$, $Val(q) \subseteq P_i$. Consider any Lamport diagram $D = (E, \leq, V, \Sigma) \in \mathcal{L}^{po}(S)$. Let ρ be an accepting run of S on D. We define the *associated model* as $M = D'$, where $D' = (E, \leq, V', 2^P)$ where $V' : E \rightarrow 2^P$ is defined as follows: For $e \in E$, $V'(e) = Val(\rho(\downarrow e)[i])$, if $e \in E_i$.

We say that an interpreted system $\mathcal{S} = (S, Val)$ *satisfies* a formula ψ of w-LTL iff $\{D' \mid D \in \mathcal{L}^{po}(S)\} \subseteq Models(\psi)$. We denote this by $S \models \psi$.

Theorem 3.2.1. *Let ψ be an w-LTL formula of length m and S be an interpreted SCA with k being the maximum of $\{Q_i \mid i \in [n]\}$. Then the question $S \models^? \psi$ can be answered in time $k^{O(n)}2^{O(mn)}$.*

Proof. To check if $S \models \psi$, we have to check if $\mathcal{L}^{po}(S) \subseteq Models(\psi)$. From the proof of Theorem 3.3.1, it follows that $S \models \psi$ iff $\mathcal{L}^{po}(S) \subseteq \mathcal{L}^{po}(S_\psi)$. But then, this is equivalent to checking if $\mathcal{L}^{po}(S) \cap \mathcal{L}^{po}(S_{\neg\psi}) = \emptyset$. We know from [66] that the class of poset languages accepted by SCAs are effectively closed under intersection and from Theorem 3.1.9 that the emptiness of the language accepted by the resulting SCA is decidable. Hence the theorem. □

3.3 Formula Automaton for w-LTL

In this section we show that one can effectively associate an SCA S_ψ with each w-LTL formula ψ in such a way that $\mathcal{L}^1(S_\psi) \neq \emptyset$ iff $TrModels(\psi) \neq \emptyset$.

Theorem 3.3.1. *Let ψ be a w-LTL formula of length m. Satisfiability of ψ over n-agent Lamport diagrams can be checked in time $2^{O(mn)}$.*

Given ψ, we first define a closure set CL_i for each $i \in [n]$. The closure set CL_i of agent i is used to define the local states of that agent and contains the i-local subformulas of ψ, their negations and some extra formulas, not present in ψ. We shall point them out in due course of time. Thereafter, we construct a local automaton for each agent i where the local transition relation non-deterministically compute the $\langle now \rangle_j$-requirements (for every $j \neq i$) and $\langle now \rangle_i$ witnesses (for every $j \neq i$) of the agent i. In this process, we guess i-local states which may be a send-to-j, receive-from-j or no-further-communication-with-j. The λ-transition, which is meant to compose the local automata, is defined in such a way that the global consistency of $\langle now \rangle_j$-requirement in agent i and corresponding $\langle now \rangle_j$-witnesses in agent j is ensured for every (i, j) pair.

As usual, we begin with the definition of sub-formula closure. We can define, for any global formula ψ, the sets of sub-formulas $CL(\psi)$ and $subf_i$ for $i \in [n]$, by simultaneous induction in such a way that:

- $\psi \in CL(\psi)$.
- if $\alpha@i \in CL(\psi)$ then $\alpha \in subf_i$.
- if $\psi' \in CL(\psi)$ then $\neg\psi' \in CL(\psi)$; a similar condition holds for $subf_i$,
  if $\neg\alpha \in subf_i$ then $\alpha \in subf_i$.
- if $\psi_1 \vee \psi_2 \in CL(\psi)$ then $\psi_1, \psi_2 \in CL(\psi)$.
- if $\beta_1 \vee \beta_2 \in subf_i$ then $\beta_1, \beta_2 \in subf_i$.
- if $\bigcirc\beta \in subf_i$ then $\beta \in subf_i$.
- if $\beta_1 \mathbf{U} \beta_2 \in subf_i$ then β_1, β_2, $\bigcirc(\beta_1 \mathbf{U} \beta_2) \in subf_i$.
- if $\diamondminus_j \beta \in subf_i$ then $\beta \in subf_j$.
- if $\langle now \rangle_j \beta \in subf_i$ then $\beta \in subf_j$.
- if $\diamondminus_j \beta \in subf_i$ then $[now]_j \diamondminus \beta \in subf_i$.

For each $i \in [n], CL_i$ contains all the elements of $subf_i$ and more. The extra elements in CL_i are needed to construct the formula automaton.

- if $\alpha \in \mathit{subf}_i$ then $\alpha \in CL_i$ too.
- if $\alpha \in CL_i$ then $\neg\alpha \in CL_i$, taking $\neg\neg\alpha$ to be α.
- $\bigcirc \mathit{True} \in CL_i$.
- for all $j \neq i$, $\mathbf{s}^j, \mathbf{r}^j, \mathbf{no\_comm}^j, \bigcirc\mathbf{s}^j, \bigcirc\mathbf{r}^j, \Box\mathbf{no\_comm}^j \in CL_i$.
- if $\langle \mathit{now} \rangle_j \alpha \in CL_i$ then $[\mathit{now}]_j \widetilde{\alpha} \in CL_i$.
  where $\widetilde{\alpha} \equiv \mathbf{no\_comm}^i \wedge (\diamondminus\alpha \vee \Diamond\alpha)$.

It can be checked that $|CL(\psi)|$ is linear in the size of ψ. For the rest of this section, fix a global formula $\psi_0 \in \Psi$. We will refer to $CL(\psi_0)$ simply as CL and CL_i will refer to the associated sets of i-local formulas. We also use $U_i \overset{\text{def}}{=} \{\alpha\mathbf{U}\beta \mid \alpha\mathbf{U}\beta \in \mathit{subf}_i\}$.

We say that $A \subseteq CL_i$ is an *i-atom* iff it satisfies the following conditions:

- for every formula $\alpha \in CL_i$, either $\alpha \in A$ or $\neg\alpha \in A$ but not both.
- for every formula $\alpha \vee \beta \in CL_i$, $\alpha \vee \beta \in A$ iff $\alpha \in A$ or $\beta \in A$.
- for every formula $\alpha\mathbf{U}\beta \in CL_i$, $\alpha\mathbf{U}\beta \in A$ iff $\beta \in A$ or $\{\alpha, \ \bigcirc(\alpha\mathbf{U}\beta)\} \subseteq A$.
- for every formula $\diamondminus\alpha \in CL_i$, if $\neg\diamondminus\alpha \in A$ then $\neg\alpha \in A$.
- for every formula $\langle \mathit{now} \rangle_i \alpha \in CL_i$, $\alpha \in A$ iff $\langle \mathit{now} \rangle_i \alpha \in A$.
- for every formula $\diamondminus_j\alpha \in CL_i$, if $\diamondminus_j\alpha \in A$ then $[\mathit{now}]_j \diamondminus\alpha \in A$.
- $\bigcirc \mathit{True} \in A$.

  this condition will ensure that every i-state is non-terminal.
- if $\mathbf{no\_comm}^j \in A$ then $\neg\bigcirc\mathbf{s}^j, \neg\bigcirc\mathbf{r}^j \in A$.
- if $\mathbf{no\_comm}^j \in A$ then $\Box\mathbf{no\_comm}^j \in A$.

  these two conditions mean that if there is no-communication-to-j in the current i-state then this will persist and that means there will not be a $\mathbf{s}^j$ or $\mathbf{r}^j$ any further.

Let AT_i denote the set of all i-atoms. Let $AT \overset{\text{def}}{=} \bigcup_i AT_i$. Let $\widetilde{AT}$ denote the set $AT_1 \times \ldots \times AT_n$. We let $\widetilde{X}, \widetilde{Y}$ *etc.*, to range over $\widetilde{AT}$, and $\widetilde{X}[i]$ to denote the i-atom in the tuple.

An n-tuple $\widetilde{X} = (A_1, \cdots, A_n)$ is atom-compatible if for every $i \neq j \in [n]$, for every $[now]_j\alpha \in CL_i$ if $[now]_j\alpha \in A_i$ then $\alpha \in A_j$.

Let ψ be a global formula. We define the notion $\psi \in \widetilde{X}$ as follows: if $\alpha \in CL_i$, then $\alpha@i \in \widetilde{X}$ iff $\alpha \in \widetilde{X}[i]$; $\neg\psi \in \widetilde{X}$ iff $\psi \notin \widetilde{X}$; $\psi_1 \vee \psi_2 \in \widetilde{X}$ iff $\psi_1 \in \widetilde{X}$ or $\psi_2 \in \widetilde{X}$.

For $i \in [n]$, let $N_i = \{\alpha \mid \langle now \rangle_i \alpha \in CL\}$. Let $N = \bigcup_i N_i$.

We will be considering a data structure that associates with each $i \in [n]$, a triple (A_i, χ_i, θ_i) where $A_i \in AT_i$, $\chi_i : ([n] - i) \to 2^{N_i}$ and $\theta_i : ([n] - i) \to 2^N$ such that for $j \neq i$, $\theta_i(j) \subseteq N_j$. Let T_i be the set of all such i-triples.

Intuitively, $\chi_i(j)$ collects *witnesses* in N_i for agent j's *requirements* of the form $\langle now \rangle_i \alpha$. Symmetrically, $\theta_i(j)$ records the $\langle now \rangle_j$ requirements of process i.

Given triples $X = (A, \chi, \theta)$ and $Y = (A', \chi', \theta')$, where $A, A' \in AT_i$, define the local relation $\rightsquigarrow_\ell$ as follows: $X \rightsquigarrow_\ell Y$ if and only if

- for every $\bigcirc\beta \in CL_i$, $\bigcirc\beta \in A$ iff $\beta \in A'$.
- for every $\diamondminus\beta \in CL_i$, if $\diamondminus\beta \in A'$ then $\beta \in A$ or $\diamondminus\beta \in A$.
- θ-update:
  - $\mathbf{s}^j, \mathbf{r}^j \notin A$: $\theta'(j) = \theta(j) \cup \{\alpha \mid \langle now \rangle_j \alpha \in A'\}$.
- χ-update:
  - Case 1: $\mathbf{s}^j \notin A$: for all $j \neq i$, $\chi'(j) = \chi(j) \cup \{\alpha \in A' \mid \langle now \rangle_i \alpha \in CL_j\}$.
  - Case 2: $\mathbf{s}^j \in A$: for all $j \neq i$, $\chi'(j) = \{\alpha \in A' \mid \langle now \rangle_i \alpha \in CL_j\}$.

$\rightsquigarrow_\ell$ is defined with keeping two issues in mind, first is the reasoning about the $\bigcirc$ and $\diamondminus_i$ modality in the proof of truth claim, whereas the second is computation of χ and β for the local automaton.

The communication constraints are defined as follows: consider triples $X = (A, \chi, \theta)$ and $Y = (B, \xi, \delta)$, where $A \in AT_i$ and $B \in AT_j$; define the communication relation $\rightsquigarrow_\lambda$ as follows. $X \rightsquigarrow_\lambda Y$ if and only if

- $\mathbf{s}^j \in A$;
- There exists $X' = (A', \chi', \theta')$, $A' \in AT_i$ such that $X \rightsquigarrow_\ell X'$;
- There exists $Y' = (B', \xi', \delta')$, $B' \in AT_j$ such that $Y' \rightsquigarrow_\ell Y$;
- $\mathbf{r}^i \in B'$;
- $\theta(j) \subseteq \xi'(i)$;
- $\theta'(j) = \{\alpha \notin \xi'(i) \mid \langle now \rangle_j \alpha \in A'\}$;
- $\delta(i) = \delta'(i) - \chi(j)$.

As already pointed out, $\rightsquigarrow_\lambda$ checks that the χ's and θ's computed locally are consistent, for the pair of agents (i, j).

Let $\mathcal{T} \subseteq T_1 \times \cdots \times T_n$ be the set of all n-triples $\widetilde{T} = \langle (A_1, \chi_1, \theta_1), \cdots (A_n, \chi_n, \theta_n) \rangle$ which are globally-compatible that is, satisfy the following conditions:

- $(A_1, \cdots, A_n)$ is atom-compatible and
- for every $i \neq j$, for every $\langle now \rangle_j \alpha \in CL_i$ if $\alpha \in \theta_i(j)$ and $\mathbf{no\_comm}^j \in A_i$ then $[now]_j \widetilde{\alpha} \in A_i$.

We call $A \in AT_i$ an initial atom iff for all $\diamondminus_i \alpha \in CL_i$, $\diamondminus_i \alpha \in A$ iff $\alpha \in A$.

We call a triple $(A, \chi, \theta) \in T_i$ initial if $\chi(j) = \{\alpha \in A \mid \langle now \rangle_i \alpha \in CL_j\}$, and $\theta(j) = \{\alpha \mid \langle now \rangle_j \alpha \in A\}$ and the atom A is initial too.

Let $\widetilde{X} \in \widetilde{AT}$. We say $\widetilde{X}$ is *initial*, iff for all i, $\widetilde{X}[i]$ is initial.

Let global state $\widetilde{Q}$ be a subset of $Q_1 \times \cdots \times Q_n$ such that an n-tuple $\langle (A_1, u_1, \chi_1, \theta_1) \cdots (A_n, u_n, \chi_n, \theta_n) \rangle \in \widetilde{Q}$ if $\langle (A_1, \chi_1, \theta_1) \cdots (A_n, \chi_n, \theta_n) \rangle \in \mathcal{T}$. We use $\widetilde{X}$, $\widetilde{Y}$, to represent members of $\widetilde{Q}$, and $\widetilde{X}(A)[i]$, $\widetilde{X}(u)[i]$ *etc.*, to denote the elements of the tuple in the i^{th} component.

We are now ready to associate an SCA with the given formula in the standard manner. For $i \in [n]$, $\Sigma_i \overset{\text{def}}{=} 2^{P_i}$ constitute the distributed alphabet over which the SCA is defined.

Definition 3.3.2. *Given any formula ψ_0, the* **SCA associated with** ψ_0 *is defined by:*

$$S_{\psi_0} \overset{\text{def}}{=} ((Q_1, G_1), \ldots, (Q_n, G_n), \rightarrow, Init)$$

where:

- $Q_i = \{(A, u, \chi, \theta) \mid (A, \chi, \theta) \in T_i, u \subseteq (U_i \cap A)\}$.
- $G_i = \{(A, u, \chi, \theta) \in Q_i \mid u = \emptyset\}$.
- $Init = \{\widetilde{X} \in \widetilde{Q} \mid \psi_0 \in (\widetilde{X}(A)[1], \ldots, \widetilde{X}(A)[n]), and\ \widetilde{X}\ is\ initial\ \}$.
- $(A, u, \chi, \theta)\xrightarrow{P'}(B, v, \chi', \theta')$, *where* $A, B \in AT_i$, *iff*

  1. $P' = A \cap P_i$.
  2. $(A, \chi, \theta) \rightsquigarrow_\ell (B, \chi', \theta')$.
  3. *The set v is defined as follows:*

$$v = \begin{cases} \{\alpha\mathbf{U}\beta \in B \mid \beta \notin B\} & \textit{if } u = \emptyset \\ \{\alpha\mathbf{U}\beta \in u \mid \beta \notin B\} & \textit{otherwise} \end{cases}$$

- $(A, u, \chi, \theta)\xrightarrow{\lambda}(B, v, \xi, \delta)$ *iff* $(A, \chi, \theta) \rightsquigarrow_\lambda (B, \xi, \delta)$.
- *For every* $i, j \in [n]$, $i \neq j$ *and* $(A, u, \chi, \theta) \in Q_i$, *if* $\mathbf{s}^j \in A$ *then there exists* $(B, v, \xi, \delta) \in Q_j$ *such that* $(A, u, \chi, \theta)\xrightarrow{\lambda}(B, v, \xi, \delta)$.

We will denote S_{ψ_0} by S_0 from now on.

Lemma 3.3.3. $\mathcal{L}^1(S_0) \neq \emptyset$ *iff* $TrModels(\psi_0) \neq \emptyset$.

Proof. Suppose $\mathcal{L}^1(S_0) \neq \emptyset$. Let $w = a_1a_2\ldots$ be an infinite word over 2^P which is accepted by $Pr^1_{S_0}$. Let $\rho = x_0x_1\ldots$ be an accepting run of $Pr^1_{S_0}$ on w. From Section 3.1.3, we know how to associate a Lamport diagram $D = (E, \leq, V)$ with ρ. There is an associated map from local states of D to instances in the run too. $\lambda : LC \rightarrow \mathbb{N}_0$ where for all $d \in LC_i$ if $d = \emptyset$ then $\lambda(d) = 0$ else if $d = \downarrow(k, k{+}1)$ for some $k \geq 0$ then $\lambda(d) = k + 1$. λ can be extended to set of all finite configurations of D, C_D^{fin}, as follows: for any $c = (d_1, \cdots, d_n) \in C_D^{fin}$, $\lambda(c) = max\{\lambda(d_1), \cdots, \lambda(d_n)\}$.

Let d be an i-local configuration of D. We associate an i-atom A_d with d as follows: for all d, $\rho(\lambda(d))[st][i]$ is a tuple (A, u, χ, θ), set $A_d = A$. Similarly, $\chi_d = \chi$, $\theta_d = \theta$ and $u_d = u$.

V is extended to LC as follows: for all $d \in LC_i$, $V(d) \overset{\text{def}}{=} (A_d \cap P_i)$.

Claim: For all $\alpha \in CL_i$, for all $d \in LC_i$, $D, d \models_i \alpha$ iff $\alpha \in A_d$.

Assuming the claim it is easy to see that $D \models \psi_0$ iff $\psi_0 \in (A_{\epsilon_1}, \ldots, A_{\epsilon_n})$. But this follows from the definition of $Init$, the set of initial states of S_0. Hence D is a model for ψ_0.

From Section 2.5.1, we know how to construct TrD from D and using the Lemma 2.5.7 we can assert that $TrD \models \psi_0$ and hence $TrModels(\psi_0) \neq \emptyset$.

We proceed to prove the claim.

Proof: The propositional and boolean cases are routine.

($\alpha = \bigcirc\beta$) Suppose $D, d \models_i \bigcirc\beta$. We must show that $\bigcirc\beta \in A_d$. By the definition of $\models_i$, there exists $d' \in LC_i$ such that $d \lessdot d'$ and $D, d' \models_i \beta$. By the definition of run, we have $\rho(\lambda(d))[i] \overset{A_d \cap P_i}{\rightarrow} \rho(\lambda(d'))[i]$ in $Pr^1_{S_0}$. Therefore, for all $\bigcirc\gamma \in subf_i$, $\bigcirc\gamma \in A_d$ iff $\gamma \in A_{d'}$. $\because$ by the induction hypothesis, $\beta \in A_{d'}$ hence, we have $\bigcirc\beta \in A_d$ and we are done.

Conversely, suppose $\bigcirc\beta \in A_d$. We must show that $D, d \models_i \bigcirc\beta$. By the induction hypothesis and by the semantics of the modality $\bigcirc$, it suffices to prove that there exists $d' \in LC_i$ such that $d \lessdot d'$ and $\beta \in A_{d'}$. Consider the state $\rho(\lambda(d))[i]$. As we observed earlier, none of the i-atoms are terminal. Hence, all of the states in Q_i are non-terminal. Therefore, $\rho(\lambda(d))[i]$ is not a terminal state. Thus, d can't be i-maximal otherwise ρ won't be good. Hence, there exists $d' \in LC_i$ such that $d \lessdot d'$. By the definition of ρ, $\rho(\lambda(d))[i] \overset{A_d \cap P_i}{\rightarrow} \rho(\lambda(d'))[i]$ in $Pr^1_{S_0}$. Therefore, $\bigcirc\beta \in A_d$ implies $\beta \in A_{d'}$. By induction hypothesis, $D, d' \models_i \beta$. Hence, we have the following: there exists $d' \in LC_i$ such that $d \lessdot d'$ and $M, d' \models_i \beta$. By the definition of $\models_i$, $M, d \models_i \bigcirc\beta$ and we are done.

($\alpha = \beta\mathbf{U}\gamma$) Suppose $D, d \models_i \beta\mathbf{U}\gamma$. We must show that $\beta\mathbf{U}\gamma \in A_d$. Since $D, d \models_i \beta\mathbf{U}\gamma$, there exists $d' \in LC_i$ such that $d \subseteq d'$, $D, d' \models_i \gamma$ and for all $d'' \in LC_i$: $d \subseteq d'' \subset d'$: $D, d'' \models_i \beta$. We show that $\beta\mathbf{U}\gamma \in A_d$ by a second induction on $l = |d'| - |d|$.

Base case: ($l = 0$).

Then, $d = d'$ and so $D, d \models_i \gamma$. By the main induction hypothesis, $\gamma \in A_d$ and (by the definition of atom), $\beta\mathbf{U}\gamma \in A_d$.

Induction step: ($l > 0$).

By the semantics of the modality $\mathbf{U}$, $D, d \models_i \beta$ and $D, d_1 \models_i \beta\mathbf{U}\gamma$ for d_1 such that $d \subset d_1 \subseteq d''$. Therefore, by the secondary induction hypothesis, $\beta\mathbf{U}\gamma \in A_{d_1}$. From the definition of $\rightarrow$, we have $\bigcirc(\beta\mathbf{U}\gamma) \in A_d$ (recall that if $\beta\mathbf{U}\gamma \in \mathit{subf}_i$ then $\bigcirc(\beta\mathbf{U}\gamma) \in \mathit{subf}_i$ as well). By the main induction hypothesis, we have $\beta \in A_d$ as well. Combining these facts and using the definition of an atom, we see that $\beta\mathbf{U}\gamma \in A_d$ as required.

Conversely, suppose $\beta\mathbf{U}\gamma \in A_d$. We must show that $D, d \models_i \beta\mathbf{U}\gamma$. Since ρ is an accepting run of $Pr^1_{S_0}$, we claim that there exists $d' \in LC_i$ such that $d \leq d'$ and $\gamma \in A_{d'}$.

Proof: Suppose not. That is, for every $d' \in LC_i$ if $d \leq d'$ then $\gamma \notin A_{d'}$. That is, for every $d' \in LC_i$ if $d \leq d'$ then $\beta\mathbf{U}\gamma \in A_{d'}$. Therefore, $\exists d^\dagger \in LC_i$, $d \leq d^\dagger$ such that for every $d'' \in LC_i$ if $d^\dagger \leq d''$ then $\beta\mathbf{U}\gamma \in u_{d''}$. That is, for every $d'' \in LC_i$ if $d^\dagger \leq d''$ then $u_{d''} \neq \emptyset$. This means $Inf_i(\rho) \cap G_i = \emptyset$. That is, ρ is not accepting, which is a contradiction. Hence, the claim.

Therefore, with such a $d' \in LC_i$ where $d \leq d'$ and $\gamma \in A_{d'}$, we do a second induction on $|d'| - |d|$ to show that $D, d \models_i \beta\mathbf{U}\gamma$.

Base case: ($(|d'| - |d|) = 0$).

Then, $d = d'$ and so $\gamma \in A_d$. Then, by the main induction hypothesis it follows that $D, d \models_i \gamma$ and so $D, d \models_i \beta\mathbf{U}\gamma$.

Induction step: ($(|d'| - |d|) > 0$).

Now, $\gamma \notin A_d$. From the definition of atoms, both β and $\bigcirc(\beta\mathbf{U}\gamma)$ must be in A_d. By the definition of $\rightarrow$, $\beta\mathbf{U}\gamma \in A_{d''}$ where $d'' \in LC_i$ such that $d \lessdot d''$. By the secondary induction hypothesis, $D, d'' \models_i \beta\mathbf{U}\gamma$. Simultaneously, by the main induction hypothesis, $D, d \models_i \beta$. Therefore, by the semantics of the modality $\mathbf{U}$, $D, d \models_i \beta\mathbf{U}\gamma$ as required.

($\alpha = \diamondminus_j\beta$) The case when ($j = i$) is easy. If $\diamondminus_i\beta \in A_d$, either $\beta \in A_d$ or we can chase the events in the past of d until we find a witness for β, or end in the initial i-atom, which has β by definition whenever it has $\diamondminus_i\beta$.

Conversely, if $D, d \models_i \diamondminus_i\beta$, then there exists $d' \in LC_i$ such that $d' \leq d$ and $D, d' \models_i \beta$. By induction hypothesis, $\beta \in A_{d'}$, and by definition of i-atom,

$\diamondminus_i\beta \in A_{d'}$. We can now use the definition of the transition $\leadsto_\ell$ to "propagate" $\diamondminus_i\beta$ to A_d.

The case when $j \neq i$ follows from the validity of the formula $\diamondminus_j\beta \equiv [now]_j\diamondminus_j\beta$ and the inductive case for the $\langle now\rangle_j$ modality being considered separately.

($\alpha = \langle now\rangle_j\beta$) Let $D, d \models_i \langle now\rangle_j\beta$. Then, by definition of $\models$, there exists a $c = (d_1, \cdots, d_n) \in C_D^{fin}$ such that $d_i = d$, $d_j = d'$ and $D, d' \models_j \beta$. We are required to show, that, $\langle now\rangle_j\beta \in A_d$. Suppose not. Then, $[now]_j\neg\beta \in A_d$ and by atom compatibility, $\neg\beta \in A'_d$. That means, by induction hypothesis, $D, d' \models_j \neg\beta$. Which is a contradiction.

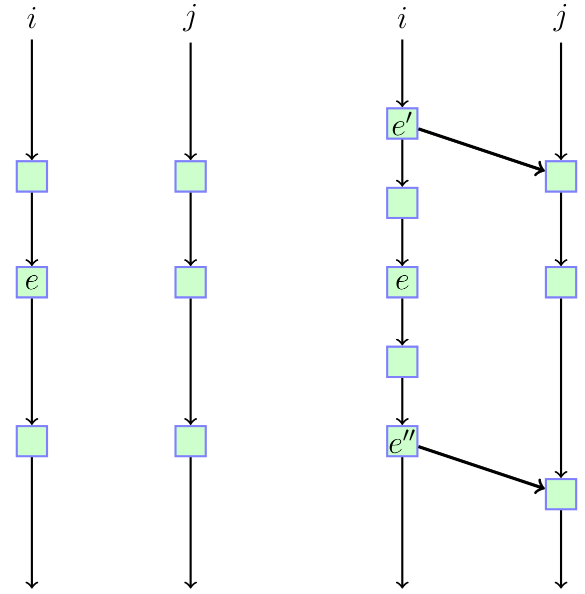
For the other direction, at the outset, we define for every Lamport diagram $D = (E, \leq, V)$, for every $e \in E_i$, for every $j \neq i$, a set $\chi(e, j)$. $\chi(e, j)$ is an interval of i-events between the send-to-j events before and after e. The interval is open or semi-open depending on the presence of send-to-j events.

$\chi(e, j) = \big(Pre_\chi(e, j) \cup Post_\chi(e, j)\big) \cap E_i$, where

- $Pre_\chi(e, j)$ is defined as follows:
  - Case 1: for all $e' \in \downarrow e$, $(\uparrow e' - \uparrow e) \cap E_j = \emptyset$; that is, there are no send-to-j events till e. Then $Pre_\chi(e, j) = \downarrow e$.
  - Case 2: Otherwise there exists $e' \in \downarrow e$ such that $(\uparrow e' \cap E_j) \neq \emptyset$ and for all $e'' \in (\downarrow e - \downarrow e')$, $(\uparrow e' - \uparrow e'') \cap E_j = \emptyset$; that is, e' is the "latest send-to-j" event before e. Then $Pre_\chi(e, j) = (\downarrow e - \downarrow e')$.
- $Post_\chi(e, j)$ is defined by:
  - Case 1: for all $e' \in \uparrow e$, $(\uparrow e - \uparrow e') \cap E_j = \emptyset$; that is, there are no "send-to-j events after e. Then $Post_\chi(e, j) = \uparrow e$.
  - Case 2: Otherwise there exists $e' \in \uparrow e$ such that $(\uparrow e - \uparrow e') \cap E_j \neq \emptyset$ and for all $e'' \in (\uparrow e' - \uparrow e)$, $(\uparrow e'' - \uparrow e') \cap E_j = \emptyset$; that is, e' is the "earliest send-to-j" event after e. Then $Post_\chi(e, j) = (\downarrow e' - \downarrow e)$.

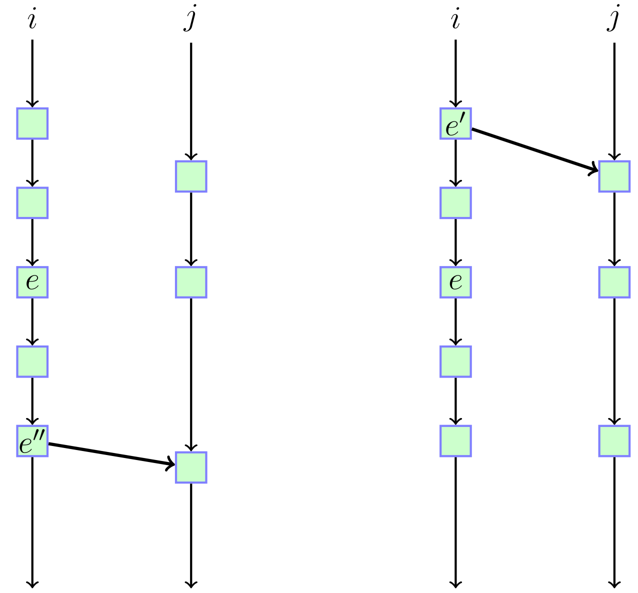
Let $C(e, j) = \chi(e, j) \cap \downarrow e$.

For every $d \in LC_i$, with the associated triple (A_d, χ_d, θ_d), $\chi_d(j)$ is computed depending on whether $d = \emptyset$ or not. As ρ is a valid run, $\rho(0)$ is an *initial*



$\chi(e, j) = \downarrow e \cup \uparrow e$ $\chi(e, j) = (\downarrow e - \downarrow e') \cup (\uparrow e - \uparrow e'')$

Figure 3.6: Lamport Diagram Fragments for $\chi(e, j)$:



$\chi(e, j) = \downarrow e \cup (\uparrow e - \uparrow e'')$ $\chi(e, j) = (\downarrow e - \downarrow e') \cup \uparrow e$

Figure 3.7: Lamport Diagram Fragments for $\chi(e, j)$:

state. Hence, $\rho(0)(A)$ is *initial* too. Therefore, by definition, $\chi_{\varepsilon_i}(j) = \{\alpha \in A_{\varepsilon_i} \mid \langle now \rangle_i \alpha \in CL_j\}$.

For the case when $d \neq \emptyset$, let $d = {\downarrow}e$ for some $e \in E_i$. Then, the following holds:

Claim 3.3.4. $\chi_d(j) = \{\alpha \in A_{d'} \mid \langle now \rangle_i \alpha \in CL_j, d' = {\downarrow}e', e' \in C(e,j)\}$.

This claim is proved by using the elaborate case analysis given in the definitions of $\chi(e,j)$ as well as the transition relation $\rightsquigarrow_\ell$.

Proof. $\chi_d(j)$, which contains $\langle now \rangle_i$ witnesses for j, is computed locally in the automaton A_{ψ_0}, and hence, depends solely on the definition of $\rightsquigarrow_\ell$. We consider the possible subcases:

(e is not a "send-to" j event) Here, again, we consider two possible subcases:

(No "send-to" j events before e) In this case $C(e,j) = {\downarrow}e \cap E_i$. Suppose $e^\dagger$ is the predecessor of e in i, that is, $e^\dagger \lessdot e$. Let $d^\dagger = {\downarrow}e^\dagger$. Clearly, $C(d^\dagger, j) = {\downarrow}e^\dagger \cap E_i = C(e,j) - \{e\}$. By induction hypothesis, $\chi_{d^\dagger}(j) = \{\alpha \in A_{d'} \mid \langle now \rangle_i \alpha \in CL_j, d' = {\downarrow}e', e' \in C(d^\dagger, j)\}$. Now, by $\rightsquigarrow_\ell$, $\chi_d(j) = \chi_{d^\dagger}(j) \cup \{\alpha \in A_d \mid \langle now \rangle_i \alpha \in CL_j\}$. Therefore, $\chi_d(j) = \{\alpha \in A_{d'} \mid \langle now \rangle_i \alpha \in CL_j, d' = {\downarrow}e', e' \in C(d,j)\}$, and we are done.

($e^\top$ is the latest "send-to" j before e) In this case, $C(d,j) = ({\downarrow}e - {\downarrow}e^\top \cup \{e^\top\}) \cap E_i$. Now, arguing exactly as above, we prove the claim.

(e is a "send-to" j event) This case is substantially more interesting. Here, $C(e,j) = \{e\}$, as the previous history of witnesses is erased. By the definition of $\rightsquigarrow_\ell$, $\chi_d(j) = \{\alpha \in A_d \mid \langle now \rangle_i \alpha \in CL_j\}$. Clearly, $\chi_d(j) = \{\alpha \in A_{d'} \mid \langle now \rangle_i \alpha \in CL_j, d' = {\downarrow}e', e' \in C(e,j)\}$, and we are done.

□

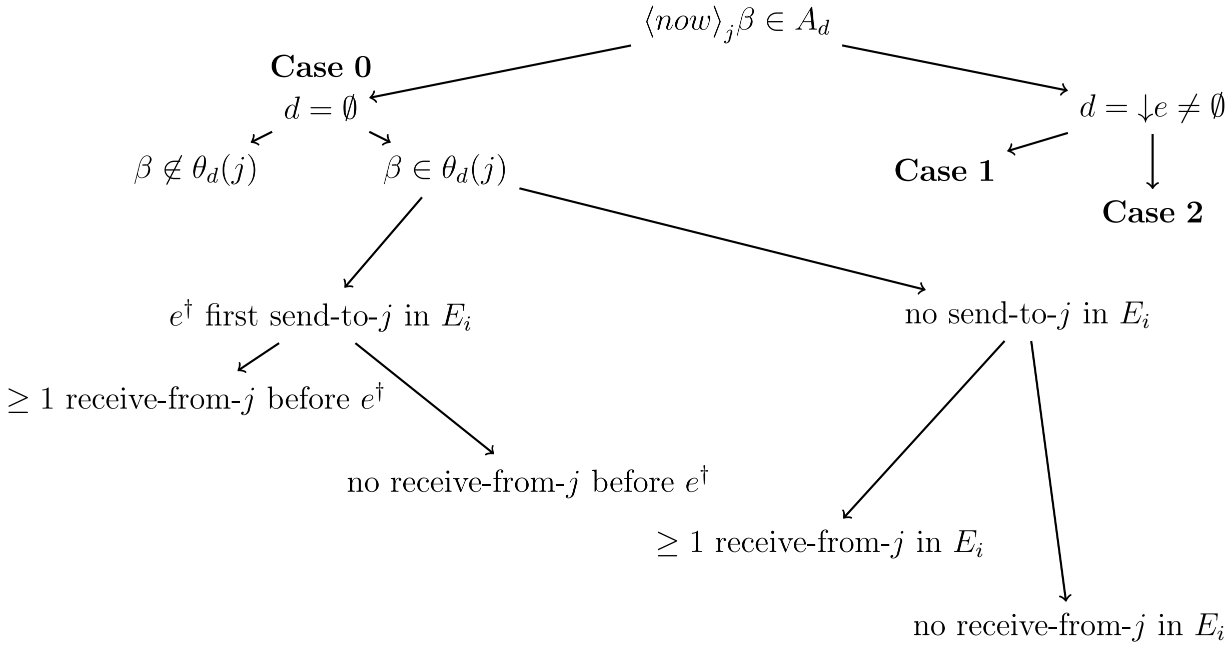

$\langle now \rangle_j \beta \in A_d$
Case 0
$d = \emptyset$
$\beta \notin \theta_d(j)$
$\beta \in \theta_d(j)$
$d = \downarrow e \neq \emptyset$
Case 1
Case 2
$e^\dagger$ first send-to-j in E_i
no send-to-j in E_i
≥ 1 receive-from-j before $e^\dagger$
no receive-from-j before $e^\dagger$
≥ 1 receive-from-j in E_i
no receive-from-j in E_i

Figure 3.8: Case Diagram for $\langle now \rangle_j \beta$

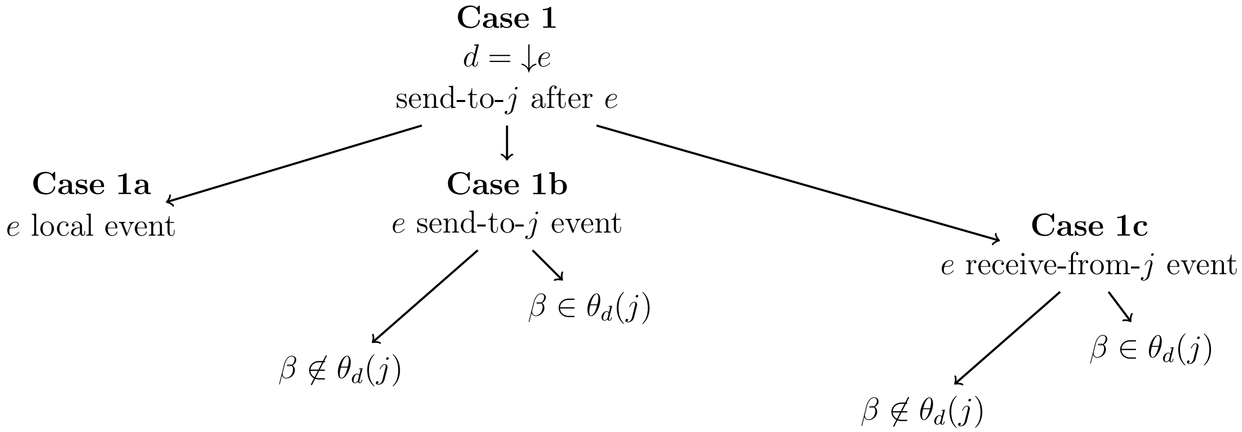


Figure 3.9: Sub-case Diagram for Case 1

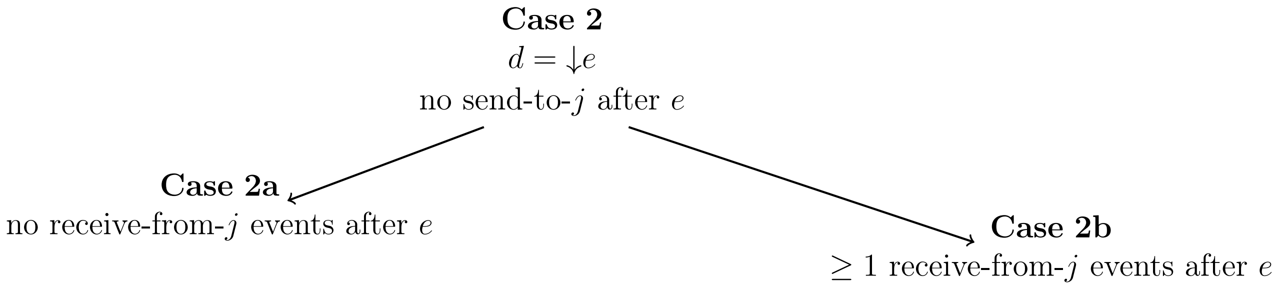

Case 2
$d = {\downarrow}e$
no send-to-j after e
Case 2a
no receive-from-j events after e
Case 2b
≥ 1 receive-from-j events after e

Figure 3.10: Sub-case Diagram for Case 2

On the other hand, $\theta_d(j)$, for $j \neq i$, contains $\langle now \rangle_j$ requirements which have to be satisfied by j. Therefore, $\theta_d(j)$ depends crucially on $\rightsquigarrow_\lambda$. Also, note that for every $j \neq i \in [n]$, $\theta_{\varepsilon_i}(j) = \{\beta \notin A_{\varepsilon_j} \mid \langle now \rangle_j \beta \in A_{\varepsilon_i}\}$

Suppose $\langle now \rangle_j \beta \in A_d$. We have to show that $D, d \models_i \langle now \rangle_j \beta$. We make the following claim:
Claim: $\exists d' \in LC_j$ *such that* $d \rangle\langle d'$ *and* $\beta \in A_{d'}$.
Assuming the claim and by induction hypothesis, we would have $\exists d' \in LC_j$ such that $d \rangle\langle d'$ and $D, d' \models_j \beta$ and, therefore, we would be done. The claim is proved by induction on $|d|$.
Proof of claim: We first consider the case when $d = \emptyset$. That is, $d = \varepsilon_i$. There are two subcases:

(Case 0a) $\beta \notin \theta_{\varepsilon_i}(j)$. By definition of $\theta_{\varepsilon_i}(j)$, $\beta \in \chi_{\varepsilon_j}(i)$. Which, in turn, implies, $\beta \in A_{\varepsilon_j}$. Also, we know, $\varepsilon_i \rangle\langle \varepsilon_j$. Therefore, given $\langle now \rangle_j \beta \in A_{\varepsilon_i}$ we have shown there exists $d' = \varepsilon_j$ such that $d \rangle\langle d'$ and $A_{d'}$.

(Case 0b) $\beta \in A_{\varepsilon_i}$. There are two subcases.

1. **There exists a send-to-j event in E_i.** Let $e^\dagger$ be the first of such send-to-j events. There can, again, be two subcases:

   (a) **There exists a receive-from-j event before $e^\dagger$.** Suppose $e_1, e_2, \cdots, e_t$, $t \geq 1$ are these receive-from-j events. There are two possibilities:

   i. There exists e_s, $1 \leq s \leq t$, such that $\beta \notin \theta_{d_s}(j)$, where $d_s = {\downarrow}e_s$. Suppose $f^\dagger$ is the corresponding send-to-i event in j. Let $d^\top = {\downarrow}f^\dagger$ and $d^\perp$ be the immediate predecessor of

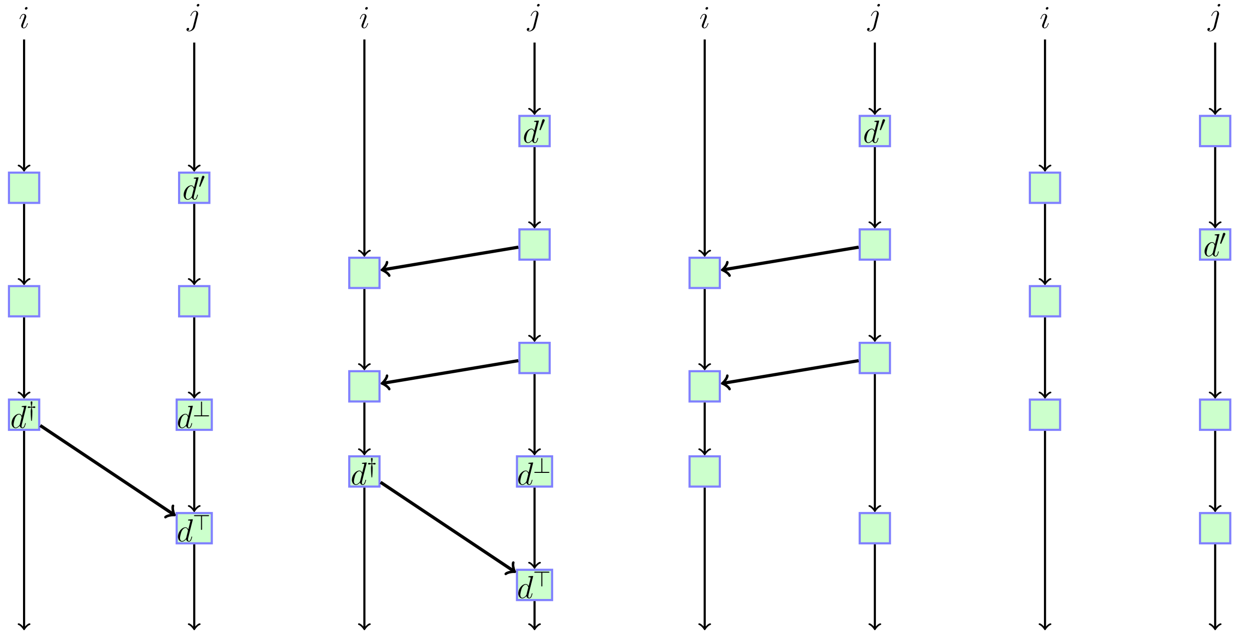


Figure 3.11: Lamport Diagram Fragments for the Case 0: $d = \varepsilon_i = \emptyset$

$d^{\top}$ in j. By the definition of $\rightsquigarrow_{\lambda}$, $\beta \in \chi_{d^{\perp}}(i)$. Clearly, this means, by the construction of $\chi_j(i)$'s, there exists $d' \in LC_j$ such that $d' \leq d^{\perp}$ and $d' \rangle\langle d$ too and $\beta \in A_{d'}$.

ii. The other case, where $\beta \in \theta_{d_s}(j)$, for every e_s, $1 \leq s \leq t$, $d_s = \downarrow e_s$, is tackled in the same way as the case where there exists no receive-from-j event in E_i before $e^{\dagger}$.

(b) **There exists no receive-from-j event before** $e^{\dagger}$. Let $d^{\dagger} = \downarrow e^{\dagger}$. Suppose $d^{\ddagger}$ be the immediate predecessor of $d^{\dagger}$. Clearly, by the construction of $\theta_i(j)$'s, $\beta \in \theta_{d^{\ddagger}}(j)$. Suppose, $f^{\dagger} \in E_j$ which is the corresponding receive-from-i. Let $d^{\top} = \downarrow f^{\dagger}$ and $d^{\perp}$ be the immediate predecessor of $d^{\top}$ in j. By the definition of $\rightsquigarrow_{\lambda}$, $\theta_{d^{\ddagger}}(j) \subseteq \chi_{d^{\perp}}(i)$. Clearly, this means, by the construction of $\chi_j(i)$'s, there exists $d' \in LC_j$ such that $d' \leq d^{\perp}$ and $d' \rangle\langle d$ too and $\beta \in A_{d'}$.

2. **There are no send-to-j event in** E_i. Here again there are two subcases.

(a) **There are receive-from-j events in** E_i. This subcase is exactly the same as $1(a)$ in **Case 0b**.

(b) **There are no receive-from-j events in** E_i. This is the case where there is no communication between i and j. Clearly, $\mathbf{no\_comm}^j \in A_d$. By compatibility property, $[now]_j\widetilde{\beta} \in A_d$ and $\widetilde{\beta} \in A_{\varepsilon_j}$. Now, $\diamondminus_j \beta \in A_{\varepsilon_j}$ or $\Diamond\beta \in A_{\varepsilon_j}$. In either case, $\exists d' \in LC_j$ such that $\varepsilon_j \leq d'$ and $\beta \in A_{d'}$. Clearly, $d'\rangle\langle\varepsilon_i$ too, and we are done.

Now, we consider the case where $d \neq \emptyset$. In order to compute $\theta_d(j)$ for each $j \neq i$, we define a set $\theta(e,j)$, where $d = \downarrow e$. $\theta(e,j) = Post_\theta(e,j) - Pre_\theta(e,j)$, where

- $Post_\theta(e,j)$ is defined as follows:
  - Case 1: there are no "send-to-j events after e. Then $Post_\theta(e,j) = E_j$.
  - Case 2: Otherwise let e' be the "earliest send-to-j" event after e. Then $Post_\theta(e,j) = \downarrow f$, where f is the j-minimum event in $\uparrow e' \cap E_j$.
- $Pre_\theta(e,j)$ is defined by:
  - Case 1: there are no "receive-from-j events till e. Then $Pre_\theta(e,j) = \emptyset$.
  - Case 2: Otherwise let e' be the "latest receive from-j" event before e. Then $Pre_\theta(e,j) = (\downarrow f - \{f\})$, where f is the j-minimum event in $\downarrow e'$.

Let $T(d,j) = \theta(e,j) \cap d$. Now, we consider two subcases in order to show that $\exists d' \in LC_j$: $d'\rangle\langle d,\ D, d' \models_j \beta$.

(Case 1) This is the case where **there is a "send-to"** j **after** e (exclusive of e). We consider three possible subcases depending on the type of e:

(Case 1a) e **is a local event**; In this case $\theta_d(j) = \theta_{d^\dagger}(j) \cup \{\beta \mid \langle now\rangle_j\beta \in A_d\}$, where $d^\dagger \in LC_i$ such that $d^\dagger \lessdot d$. Clearly, by definition, $\beta \in \theta_d(j)$. Let $e^\top \in E_i$ be the "send-to" j event after e. Let $d^\top \in LC_i$ such that $d^\top \lessdot \downarrow e^\top$. $\beta \in \theta_{d^\top}(j)$ too. Let $f \in E_j$ be the corresponding "receive-from" i event. Let $f^\perp \in E_j$ such that $f^\perp \lessdot f$ and $d^\perp = \downarrow f^\perp$. $\because$, ρ is a valid run and $e^\top \lessdot f$, $\therefore$ by the definition of $\rightsquigarrow_\lambda$, $\theta_{d^\top}(j) \subseteq \chi_{d^\perp}(i)$. Therefore, $\beta \in \chi_{d^\perp}(i)$ too. By the construction of $\chi_{d^\perp}(i)$, we can assert $\beta \in A_{d'}$ where $d\rangle\langle d'$ and we are done.

(Case 1b) e **is a send-to** j **event**; In this case $\theta_d(j) = \{\beta \notin A_{d'} \mid \langle now\rangle_j \beta \in A_d, d' = \downarrow f', f' \in T(d,j)\}$. There are two possibilities here.

$\beta \notin \theta_d(j)$**:** This means $\exists d' = \downarrow f'$, $f' \in \theta(e,j) \cap \downarrow e$ such that $\beta \in A_{d'}$. Clearly, by definition of $T(d,j)$, $d\rangle\langle d'$ and we are done.

$\beta \in \theta_d(j)$**:** This case is tackled exactly as the case where e is a local event.

(Case 1c) e **is a receive-from** j **event**; In this case $\theta_d(j) = \{\beta \notin A_{d'} \mid \langle now\rangle_j \beta \in A_d, d' = \downarrow e', e' \in \theta(e,j) \cap \downarrow e\}$. Here, again,there are two possibilities.

$\beta \notin \theta_d(j)$**:** This means, that for $d' = \downarrow f'$ where $f' \lessdot e$ and $f' \in E_j$ such that $\beta \in A_{d'}$ as $T(d,j)$ is a singleton $\{f'\}$. Clearly, $d\rangle\langle d'$ and we are done.

$\beta \in \theta_d(j)$**:** This case is tackled exactly as the case where e is a local event.

(Case 2) This is the case where **there is no "send-to"** j **after** e. We consider two subcases:

(Case 2a) Consider the case when **there is no "receive-from"** j **after** e (excluding e). We are given, that, $\langle now\rangle_j \beta \in A_d$. Clearly, in this case, $\textbf{no\_comm}^j \in A_d$. Then, $[now]_j \Diamond \widetilde{\beta} \in A_d$. Let $d^\ddagger \in LC_j$ be the j-minimal in d. By the global compatibility, $\widetilde{\beta} \in A_{d^\ddagger}$. By a separate induction and using the Büchi condition for $\mathbf{U}$, we can show $\exists d' \in LC_j$, $d^\ddagger \leq d'$ such that $\beta \in A_{d'}$. But $d\rangle\langle d'$ too. Hence, we have $\exists d' \in LC_j$ such that $d\rangle\langle d'$ and $\beta \in A_{d'}$ and we are done.

(Case 2b) Consider the case where **there is a "receive-from"** j **after** e (excluding e). Clearly, in this case $\textbf{no\_comm}^j \notin A_d$. So, we have $\beta \in \theta_d(j)$. At each $e^\dagger \in E_i$ which is a "receive-from" j $\theta_{d^\dagger}(j)$ is modified. There are two possibilities:

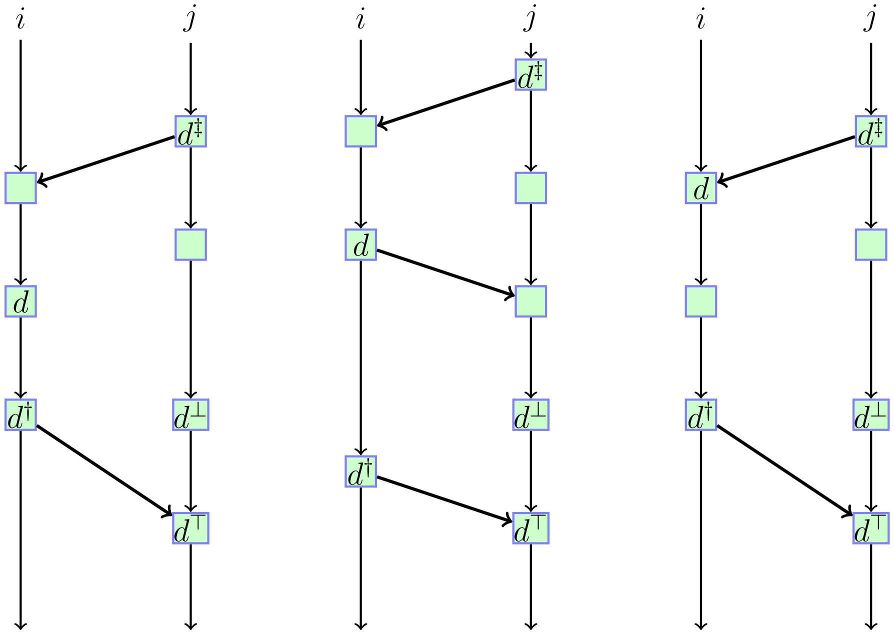


Figure 3.12: Case 1: $d = \downarrow e$; $e^{\dagger}$ closest send-to-j after e

$\exists d^{\top} \in LC_i$ such that $d < d^{\top}$, $d^{\top}$ is a "receive-from" j and $\beta \notin \theta_{d^{\top}}$. This means, $\exists d' \in LC_j$ such that $\beta \in \chi_{d'}(i)$ and, by construction of $\chi(i)$'s $d^{\top}\rangle\langle d'$. Note, that, $d\rangle\langle d'$ too.

Consider the other possibility. Let $d^{\dagger}$ be the last such "receive-from" j and $\alpha \in \theta_{d^{\dagger}}(j)$. Clearly, $\mathbf{no\_comm}^j \in A_{d^{\dagger}}$. Now, we argue as for the Case 2a.

Thus $D, d \models_i \langle now\rangle_j \beta$ iff $\langle now\rangle_j \beta \in A_d$.

Conversely, suppose $TrD \models \psi_0$, where $TrD = (E, \leq, V)$. To show that TrD is a member of $\mathcal{L}^1(S_0)$, we have to construct an accepting run of $Pr^1_{S_0}$ on TrD. For any i-local configuration d of TrD, let $\nu_i(d) \overset{\text{def}}{=} \{\alpha \in CL_i \mid TrD, d \models_i \alpha\}$.

Let $i \in [n], A_i(\emptyset) = A_i^0$. Define $\chi_i^0(j) = \{\alpha \in A_i \mid \langle now\rangle_i \alpha \in CL_j\}$, and $\theta_i^0(j) = \{\alpha \mid \langle now\rangle_j \alpha \in A_i\}$. Let $\widetilde{q}_0 = (A_1(\emptyset), \emptyset, \chi_1^0, \theta_1^0), \ldots, (A_n(\emptyset), \emptyset, \chi_n^0, \theta_n^0)$ be the initial state and $Bf_{\epsilon} \in \mathcal{B}$ be the initial buffer contents.

We can compute χ_i's independent of θ_i's. This is done inductively over the size of $d \in LC_i$.

$\chi_i^{\varepsilon_i}(j) = \{\alpha \in A_i^{\varepsilon_i} \mid \langle now\rangle_i \alpha \in CL_j\}$.

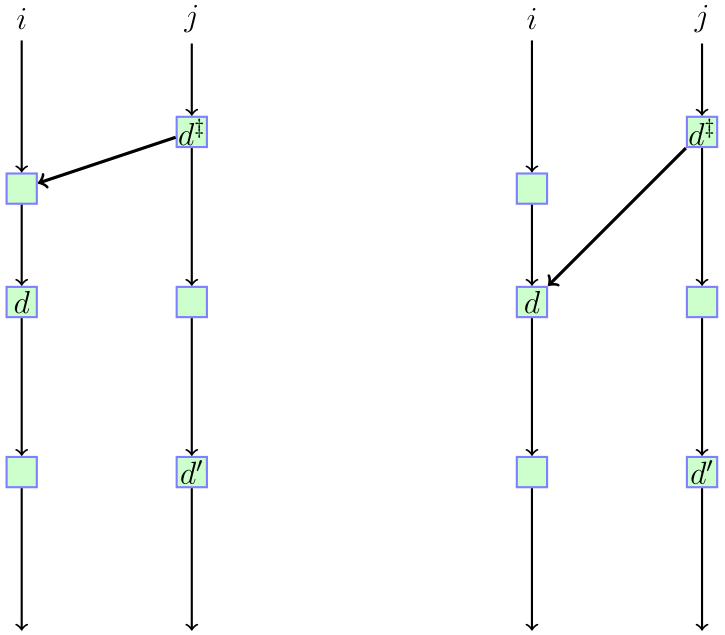


Figure 3.13: Case 2a: $d = {\downarrow}e$; no send-to-j after e

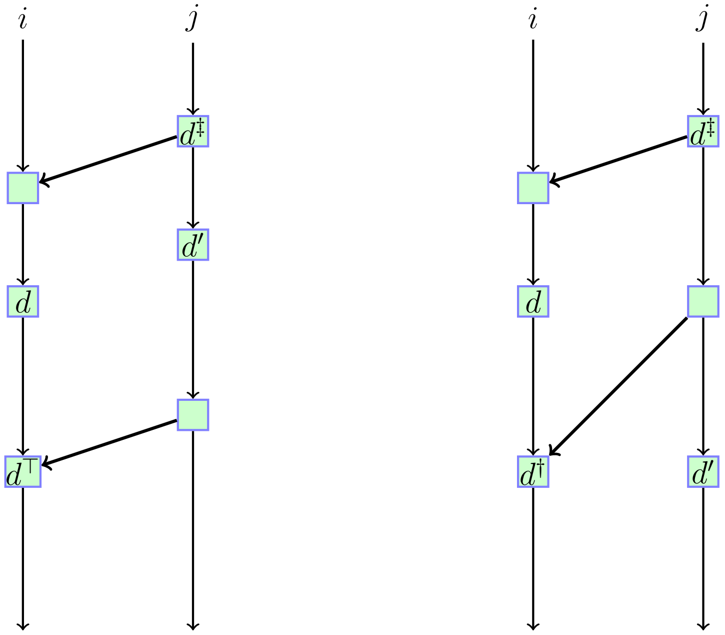


Figure 3.14: Case 2b: $d = {\downarrow}e$; no send-to-j after e

Suppose, we have computed $\chi_i^{d'}(j)$, for some $d' \in LC_i$; $|d'| \geq 0$. Let $d = \downarrow e \in LC_i$ such that $d \lessdot d'$. For each $j \neq i$, $\chi_i^d(j)$ is computed as follows:

$$\chi_i^d(j) = \begin{cases} \{\alpha \in A_i(d) \mid \langle now \rangle_i \alpha \in CL_j\}, & \text{if } e \text{ is a send-to-} j \text{ event.} \\ \chi_i^{d'}(j) \cup \{\alpha \in A_i(d_i) \mid \langle now \rangle_i \alpha \in CL_j\}, & \text{if } e \text{ is } \texttt{not} \text{ a send-to-} j \text{ event.} \end{cases}$$

Let $\sigma = e_1 e_2 \cdots e_k e_{k+1} \cdots$ be a 1-bounded linearization of TrD. We define a map $\rho : prefix(\sigma) \to \mathcal{Q}$, from prefixes of σ to the set of global states of $Pr^*_{S_0}$ inductively as follows:

$\rho(\varepsilon) = (\widetilde{q}_0, Bf_\epsilon)$.

Inductively, let $e_1 \cdots e_k = \sigma^k$ be the k-length prefix of σ for which $\rho(\sigma^k)$ is defined as $((\nu_1^k, u_1^k, \chi_1^k, \theta_1^k), \ldots, (\nu_n^k, u_n^k, \chi_n^k, \theta_n^k), Bf_k)$. Let $\sigma^{k+1} = \sigma^k \cdot e_{k+1}$. We have to compute $\rho(\sigma^{k+1}) = ((\nu_1^{k+1}, u_1^{k+1}, \chi_1^{k+1}, \theta_1^{k+1}), \ldots, (\nu_n^{k+1}, u_n^{k+1}, \chi_n^{k+1}, \theta_n^{k+1}), Bf_{k+1})$ where for $j \neq i$, $d'_j = d_j$, $u_j^{k+1} = u_j^k$. Let $A = \nu_i(d_i)$ and $B = \nu_i(d'_i)$ where $d'_i = \downarrow e_k$. If $u_i^k = \emptyset$ then $u_i^{k+1} = \{\alpha \mathbf{U} \beta \in B \mid \beta \notin B\}$; otherwise, $u_i^{k+1} = \{\alpha \mathbf{U} \beta \in u_i^k \mid \beta \notin B\}$.

We look at two cases:

1. e_k **is an i-local event**; There may be two cases to be considered while updating requirement set $\theta_i(j)$, for each $j \neq i$.

   no_comm$^j \in A_i^k$**:** In this case $\langle now \rangle_j$ requirements are updated in the same way as we update until requirements.

   $$\theta_i^{k+1}(j) = \begin{cases} \{\beta \notin \chi_j^k(i) \mid \beta \in \theta_i^k(j)\} & \theta_i^k(j) \neq \emptyset \\ \{\beta \notin \chi_j^k(i) \mid \beta \in A_i^{k+1}\} & \theta_i^k(j) = \emptyset \end{cases}$$

   no_comm$^j \notin A_i^k$**)** In this case requirements could simply be added up as follows: $\theta_i^{k+1}(j) = \theta_i^k(j) \cup \{\beta \mid \langle now \rangle_j \beta \in A_i^{k+1}\}$.

   In the case of e being local event buffers don't change. Therefore, for all $i \neq j \in [n]$, $Bf'(i,j) = Bf(i,j)$.

2. e_k **is not an i-local event**; Here again, there could be two possible subcases:

   (a) e_k is send-to-j event in agent i. Let f be the corresponding receive-from-i event in j. Let $d^\dagger$ be the immediate predecessor of $\downarrow f$. Then, the update takes place as follows:

$\theta'_i(j) = \{\alpha \not\in \chi_j^{d^\dagger}(i) \mid \langle now \rangle_j \alpha \in A_i(d'_i)\}$.

Let $\widetilde{q}$. In the case of send-to-j we insert into the (i,j)th point. That is, $Bf_{k+1} = \mathbf{insert}Bf_k, (i, j, \widetilde{q}_i^k)$.

(b) e_k is receive-from-i event in agent j. Let f be the corresponding send-to-j event in i. Suppose $f = e_l$ in σ. Let $d^\dagger$ be the immediate predecessor of $\downarrow f$. Then, the update takes place as follows:

$\theta'_j(i) = (\theta_i(j) - \chi_i^{d^\dagger}(j)) \cup \{\alpha \not\in \chi_i^{d^\dagger}(j) \mid \langle now \rangle_j \alpha \in A_j(d'_j)\}$.

In the case of receive-from-i we delete from the (i,j)th point. That is, $Bf_{k+1} = \mathbf{insert}Bf_k, (i, j, \widetilde{q}_i^l)$.

For all $j \neq i$, for all $j' \neq j$, $\chi_j(j') = \chi'_j(j')$ and $\theta'_j(j') = \theta_j(j')$.

It is now easily shown that ρ is an accepting run of $Pr^1_{S_0}$ on TrD. and hence that $TrD \in \mathcal{L}^1(S_0)$. □

From the above lemma, it follows that deciding satisfiability of ψ_0 amounts to checking emptiness of the SCA $Pr^1_{S_0}$. From Theorem 3.1.9, it follows that emptiness of the poset language accepted by an SCA can be checked in time $k^{O(n)}$, where k is the maximum of $\{|Q_i| \mid i \in [n]\}$. Now the time bound stated in Theorem 3.3.1 follows by observing that each component in $Pr^1_{S_0}$ has a maximum of $2^{O(m)}$ states, where m is the size of ψ.

4

Client-Server Systems with Unbounded Agents

One of the most important challenges in algorithmic verification is to extend this technique to infinite-state systems. There can be, broadly, two reasons why a program may have infinite state space. Firstly, it may operate on unbounded data structures. Examples of such systems include timed automata [5], data-independent systems [85], relational automata [21], pushdown processes [20] and lossy channel systems [2]. Another reason is that the program may have an infinite control part. This is the case in Petri Nets [33], [51] and parameterized systems. In the latter, topology of the system is parameterized by the number of processes in the system [36], [3], [52], [26], [56], [87] and we want to prove the correctness of the system regardless of the number of processes.

One major approach is to extend the paradigm of symbolic model checking [19] to new classes of models by an appropriate symbolic representation. Regular model checking is an extension in which the state and transition relations are represented by regular sets, typically over finite or infinite words or tree structures. Most of the research work has focussed on models where configurations can be represented as finite words of arbitrary length over a finite alphabet. Regular model checking was advocated by Kesten *et. al.* [52] and by Boigelot and Wolper [86] as a uniform framework for analyzing several classes of parameterized and infinite-state systems. The idea is that regular sets provide an efficient representation of infinite spaces and play a role similar to that by Binary Decision Diagrams (BDDs) for symbolic model checking of finite state systems. One can also exploit automata-theoretic algorithms for manipulating regular sets. Such algorithms have been successfully implemented, *e.g.*, in the MONA [42] system.

We approach the verification of infinite-state system in a different way. Given the system description in some finite representation and the specified safety/liveness property in some logic, we argue that most of these interesting specifications can be written in a logical language where the formulas, owing to their inbuilt syntactic constraints, give a bound on the system in question. Therefore, consequently, the infinite verification problem reduces to the standard verification problem over finite states [28].

In this chapter we describe two automaton models for client-server systems and show that they are equivalent to multi-counter automata. We also give one example each for these systems thereby illustrating their use in modelling real life cases. In a subsequent chapter, we propose two logics, respectively, for each class, and show how standard formula automaton techniques can be used in this setting.

4.1 Automata Models for Client-Server Systems

We consider client-server systems of two types, known as discrete and session-oriented in the literature [22]. In the first case, the clients simply send requests and wait for the responses (either yes or no) from the server. We call them client-server systems with passive clientele. In the other case, there is non-trivial interaction between the client and the server between the send-request and the receive-response. We call these client-server systems with active clientele.

Fix CN, a countable set of *client names.* In general, this set would be recursively generated using a naming scheme, for instance using sequence numbers and time-stamps generated by processes. We choose to ignore this structure for the sake of technical simplicity. We use a, b *etc.* with or without subscripts to denote elements of CN.

4.1.1 Passive Clients

Fix Γ_0, a finite service alphabet. We use u, v *etc.* to denote elements of Γ_0, and they are thought of as types of services provided by a server. An *extended alphabet* is a set $\Gamma = \{req_u, ans_u \mid u \in \Gamma_0\} \cup \{\tau\}$. These refer to requests for such service and answers to such requests, as well as "silent" internal action τ.

Elements of Γ_0 represent logical types of services that the server is willing to provide. This means that when two clients ask for a service of the same type, given by an element of Γ_0 it can tell them apart only by their name. We could in fact then insist that server's behaviour be identical towards both, but we do not make such an assumption, to allow for generality.

We define below systems of services that handle passive clientele. Servers are modelled as state transition systems which identify clients only by the type of service they are associated with. Thus, transitions are associated with client types rather than client names.

Definition 4.1.1. *A* ***Service for Passive Clients*** *(SPS) is a tuple* $M = (S, \delta, I, F)$ *where* S *is a finite set of states,* $\delta \subseteq (S \times \Gamma \times S)$ *is a server transition relation,* $I \subseteq S$ *is the set of initial states and* F *the set of final states of* M*.*

Without loss of generality we assume that in every SPS, the transition relation δ is such that for every $s \in S$, there exists $r \in \Gamma$ such that for some $s' \in S$, $(s, r, s') \in \delta$. The use of silent action τ makes this an easy assumption.

Note that an SPS is a finite state description. A transition of the form (s, req_u, s') refers implicitly to a new client of type u rather than to any specific client name. The meaning of this is provided in the run generation mechanism described below.

A configuration of an SPS M is a triple (s, C, χ) where $s \in S$, C is a finite subset of CN and $\chi : C \to \Gamma_0$. Thus a configuration specifies the control state of the server, as well as the finite set of *active* clients at that configuration and their types.

We use the convention that when $C = \emptyset$, the graph of χ is the empty set as well. Let Ω_M denote the set of all configuration of M; note that it is this *infinite* configuration space that is navigated by behaviours of M. A configuration (s, C, χ) is said to be *initial* if $s \in I$ and $C = \emptyset$.

We can extend the transition relation δ to configuration $\stackrel{r}{\Longrightarrow} \subseteq (\Omega_M \times \Gamma \times \Omega_M)$ as follows: $(s, C, \chi) \stackrel{r}{\Longrightarrow} (s', C', \chi')$ iff $(s, r, s') \in \delta$ and the following conditions hold:

- when $r = \tau$, $C = C'$ and $\chi = \chi'$;
- when $r = req_u$, $C' = C \cup \{a\}$, $\chi'(a) = u$ and $\chi' \lceil C = \chi$, where a is the least element of $CN - C$;

- when $r = ans_u$, $X = \{a \in C \mid \chi(a) = u\} \neq \emptyset$, $C' = C - \{a\}$ where a is the least in the enumeration of X, and $\chi' = \chi\lceil C'$.

A **run** of an SPS M is an infinite sequence of configurations $\rho = c_0 r_1 c_1 \cdots r_n c_n \cdots$, where c_0 is initial, and for all $j > 0$, $c_{j-1} \overset{r}{\Longrightarrow} c_j$. Let R_M denote the set of all runs of M.

Note that runs have considerable structure. For instance, the configuration space Ω_M can have an infinite path generated by a self-loop of the form (s, req_x, s) in δ which corresponds to an infinite sequence of service requests of a particular type. Thus, these systems have interesting reachability properties. But, as we shall see, our main use of these systems are as models of a temporal logic, and since the logic is rather weak, information present in the runs will be under-utilized.

4.1.2 Active Clients

We now consider clients who interact with the server in some non-trivial fashion. Towards this, we fix a finite **interaction alphabet** Θ. In addition, we need to specify abstract client names in the server transition system, hence we fix a finite **abstract name alphabet** Π. The service alphabet is now given by: $\Gamma = (\Theta \times \Pi \times \{0, 1\}) \cup \{\tau\}$. We further assume that there is a map $\lambda : \Pi \rightarrow \Gamma_0$ that uniquely identifies the service type associated with every client.

Note that the client's behaviour evolves temporally and the server needs to keep track of changes, and hence the clients' identity needs to be recorded. But if we do this we will have an infinite alphabet labelling such systems. To avoid this, we use the same technique as for passive clients, generating names on-the-fly. However, we do need to match client names within the transition system, so we add information about when a type is associated with a *new* name. Thus, we have a third component in the service alphabet Γ, where 1 denotes a new name to be generated and 0 refers to an existing agent.

Definition 4.1.2. *A **Service for Active Clients** (SAS) is a tuple $M = (S, \delta, I, F)$ as in the case of SPSs, with S a finite set of states, the server transition relation $\delta \subseteq (S \times \Gamma \times S)$, $I \subseteq S$ the set of initial states and $I \subseteq S$ the set of final states of M. In addition, for each $u \in \Gamma_0$, we have a **Client Transition System** $M_u = (Q_u, \delta_u, I_u, F_u)$ where Q_u is a finite set, $\delta_u \subseteq (Q_u \times \Theta \times Q_u)$, $I_u \subseteq Q_u$ and $F_u \subseteq Q_u$.*

As before, a **configuration** of an SAS M is a 4-tuple (s, C, χ, π) where $s \in S$, C is a finite subset of CN, but now $\chi : C \rightarrow Q$, where $Q = \bigcup_{u \in \Gamma_0} Q_u$, and $\pi : \Pi \rightarrow C$. Remember that a single configuration is essentially a global state of the given SAS M.

Let Ω_M denote the set of all configurations of M. As before, a configuration (s, C, χ, π) is said to be *initial* if $s \in I$ and $C = \emptyset$. The extended transition relation on configurations is defined as follows: $\overset{r}{\Longrightarrow} \subseteq (\Omega_M \times \Gamma \times \Omega_M)$ such that $(s, C, \chi, \pi) \overset{r}{\Longrightarrow} (s', C', \chi', \pi')$ iff $(s, r, s') \in \delta$ and the following conditions hold:

- when $r = \tau$, $C = C'$ and $\chi = \chi'$.

  Clearly, in the case of internal (silent) transition the set of active clients and their corresponding states don't change.

- when $r = (\theta, x, 0)$, $C' \subseteq C$ and there exists $a \in C$ such that $\pi(x) = a$, $u = \lambda(x)$, and

  – $(\chi(a), \theta, \chi'(a)) \subseteq \delta_u$, when $a \in C'$.

  – $(\chi(a), \theta, q) \subseteq \delta_u$, for some $q \in F_u$ when $a \notin C'$.

  When an already active client, referred to by x, makes a θ transition then it is registered by the main server and if it drops out in the subsequent state then it is made sure that the local client state was a final one. It is not necessary that a client with a final state as target state always gets removed from the set of active states. We just make sure that the removal was not done from an inappropriate state.

- when $r = (\theta, x, 1)$, $C' = C \cup \{a\}$ where a is the least element of $CN - C$; $\chi' \lceil C = \chi$; $(q, \theta, \chi'(a)) \subseteq \delta_u$, where $u = \lambda(x)$ and $q \in I_u$.

  When a fresh client is admitted into the system it is made sure that the source state of the local client θ transition is an initial state.

Also, note, that in a transition it is not necessary that the source valuation π and target valuation π' match. Thus, the machine can remember two different clients via x in the source and target configurations.

Coming to runs of SAS, they are defined exactly as in the case of SPS. That is, a **run** of an SAS M is an infinite sequence of configurations $\rho = c_0 r_1 c_1 \cdots r_n c_n \cdots$, where c_0 is initial, and for all $j > 0$, $c_{j-1} \xRightarrow{r} c_j$. Similarly, we let R_M denote the set of runs of M.

Note, that the definition of of a run hides some detail. In general, an interaction of the form (θ, x, ν) means that the client need not know its own identity. Rather, the server picks a new identity a from CN and uses it to keep track of messages from that client. Similarly Q_u does not indicate the actual state of a client of type u but merely the state information as recorded by the server about a client whose type is u.

Operationally, the SAS uses abstract client names that get instantiated at run time, using the flag ν, and the use of the same abstract name in different transitions is for matching interactions with the same client. This adds considerable expressive power in terms of system behaviours.

4.2 Decision Algorithms for SPS/SAS

In systems modelled by transition systems, like ours, most of the system analysis problems reduce to various kind of reachability problems on these models [15]. The system analysis therefore needs algorithms that compute the set of all *predecessors* and/or *successors* of a given set of configurations (states) S.

Let $pre(S)$ denote the set of immediate predecessors of S and $post(S)$ denote the set of immediate successors of S (via a single transition). Also, let $post^*(S)$ and $pre^*(S)$ denote the set if all its successors and predecessors. Clearly, $post^*(S)$ is the limit of the *infinite* non-decreasing sequence $(X_i)_{i \geq 0}$ given by $X_0 = S$ and $X_{i+1} = X_i \cup post(X_i)$ for every $i \geq 0$. Similarly, $pre^*(S)$ is the limit of the infinite sequence obtained by considering the *pre* function instead of the *post*.

From these definitions, one can derive straightforwardly iterative procedures for computing the $post^*(S)$ and $pre^*(S)$ that consists simply of computing the elements of the sequence of the X_i's and checking for each index i, whether $X_{i+1} = X_i$, in which case we know that the limit is reached.

For systems modelled as finite-state automata, such algorithms are guaranteed to terminate. However for SPS/SAS, the sets X_i are in general infinite and the sequence $(X_i)_{i \geq 0}$ is not guaranteed to converge in a finite number of steps.

In order to verify such systems we need to find a class of *finite structures* that can represent the infinite set of states we are interested in. Moreover in order to apply the forward/backward reachability analysis algorithms, this class of structures should at least be effectively closed under union, intersection, and the *post* and *pre* operators. Finally, since we have to compare two sets to detect convergence and we wish to check whether a set is empty, the equality and emptiness problems of this class of structures should be decidable.

4.2.1 Multi-counter Automata

Multi-counter automata (MCA) without zero-test seem to be an appropriate formal model to reason about reachability in SPS/SAS. They are known as Vector Addition Systems with States (VASS) in the literature [48]. Multi-counter automata model communicating systems through unbounded buffers when the ordering between messages in the FIFO communication channels is not relevant but only their number. They are equivalent to pure Petri Nets with decidable reachability [55] [58] [65]. They are seen to be closed under union and intersection but not complementation.

First, we define one-counter automata.

Definition 4.2.1 (One-counter automata). *A One-counter automaton is a tuple* $\mathcal{A} = (Q, q_I, F, \delta)$ *where*

- Q *is a finite set of control locations (states),*
- $q_I \in Q$ *is the initial control location (state),*
- $F \subseteq Q$ *is the set of accepting locations (states) and*
- $\delta \subseteq Q \times L \times Q$ *is the transition relation over the instruction set* $L = \{\mathbf{inc}, \mathbf{dec}, \tau\}$. *A transition labelled with* **inc** *denotes an increase (by 1) in the counter, while one labelled with* **dec** *denotes a decrease (by 1) in the counter and one labelled with* τ *is a "silent" transition which leaves the counter as it is.*

A counter valuation n is an element of $\mathbb{N}$ and a configuration of $\mathcal{A}$ is a pair in $Q \times \mathbb{N}$. The initial configuration is the pair $(q_I, 0)$. A one-counter automaton $\mathcal{A}$ induces a (possibly infinite) transition system $(Q \times \mathbb{N}, \rightarrow)$ such that $(q, n) \rightarrow (q', n')$ iff one of following holds:

- $(q, \mathbf{inc}, q') \in \delta$ and $n' = n + 1$,
- $(q, \mathbf{dec}, q') \in \delta$ and $n' = n - 1$ and $n > 0$,
- $(q, \tau, q') \in \delta$ and $n' = n$.

A finite run ρ is a finite sequence $\rho = (q_0, n_0) \to (q_1, n_1) \to \cdots (q_k, n_k)$ where (q_0, n_0) is an initial configuration. On the same lines, we can define infinite runs. Let $R_{\mathcal{A}}$ be the set of all valid runs of a given one-counter automaton $\mathcal{A}$.

We can easily extend the above definition to multi-counter case as follows:

Definition 4.2.2 (Multi-counter automata)**.** *A k-counter automaton, for $k > 0$, is a tuple $\mathcal{A} = (Q, q_I, F, \delta)$ with Q, q_I, F as in one-counter automaton and the transition relation $\delta \subseteq Q \times L \times Q$ is defined over extended set of instructions $L = \{\mathbf{inc}_1, \cdots, \mathbf{inc}_k, \mathbf{dec}_1, \cdots, \mathbf{dec}_k, \tau\}$.*

In the multi-counter case, we have $\mathbf{inc}_i$ and $\mathbf{dec}_i$ labels corresponding to each counter $1 \leq i \leq k$, whereas the "silent action" τ captures the case when none of the counters is modified.

In a k-counter automaton $\mathcal{A}$, a counter valuation is a k-tuple $(n_1, \cdots, n_k) \in \mathbb{N}^k$ and a configuration of $\mathcal{A}$ is a $(k+1)$-tuple $(q, n_1, \cdots, n_k) \in Q \times \mathbb{N}^k$. The counter valuation $(n_1, \cdots, n_k)$ may be abbreviated as $\widetilde{n}$ such that for all $1 \leq j \leq k$, $\widetilde{n}[j] = n_j$. The initial configuration is $(q_I, \widetilde{n}^\dagger)$, where $\forall j$, $1 \leq j \leq k$, $\widetilde{n}^\dagger[j] = 0$. A k-counter automaton $\mathcal{A}$ induces a (possibly infinite) transition system $(Q \times \mathbb{N}^k, \to)$ as follows: $(q, \widetilde{n}) \to (q', \widetilde{n}')$ iff one of the following holds:

- $(q, \mathbf{inc}_j, q') \in \delta$ and $\widetilde{n}'[j] = \widetilde{n}[j] + 1$, Also, for all $j' \neq j$, $\widetilde{n}[j'] = \widetilde{n}[j']$.
- $(q, \mathbf{dec}_j, q') \in \delta$ and $\widetilde{n}'[j] = \widetilde{n}[j] - 1$ and $\widetilde{n}'[j] > 0$. Also, for all $j' \neq j$, $\widetilde{n}[j'] = \widetilde{n}[j']$.
- $(q, \tau, q') \in \delta$ and for all $1 \leq j \leq k$, $\widetilde{n}'[j] = \widetilde{n}[j]$.

We can define runs of multi-counter automata as we have done for one-counter automata. Let $R_{\mathcal{A}}$ be the set of all valid runs of multi-counter atomaton $\mathcal{A}$.

We would like to show that behaviours of the class of SAS machines is equivalent to multi-counter automata. This is accomplished by encoding an SAS into a multi-counter machine and vice versa.

At the outset we observe the following. Any k-client SPS M can be modified to a k-client SAS M' such that runs of M and M' coincide.

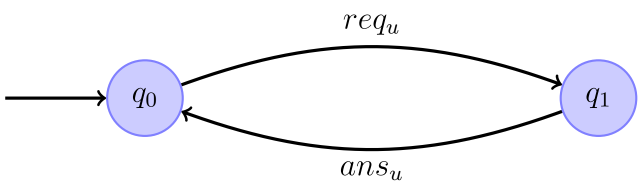


Figure 4.1: The u-type Client Transition System for SPS

4.2.2 Encoding SPS into SAS

Let there be an SPS $M = (S, \delta, I)$ with $|\Gamma_0| = k$. We define a k-client SAS $M' = (S', \delta', I')$ such that $R_M = R_{M'}$. For M' the interaction alphabet is $\Theta = \{req_u, ans_u \mid u \in \Gamma_0\}$ and abstract name alphabet is $\Pi = \{x_u \mid u \in \Gamma_0\}$. The client machines are as given in the Figure 4.1. The server machine is defined as follows:

- $S' = S$, $I' = I$,
- $\delta' \subseteq S' \times \Gamma \times S'$ is the smallest set satisfying the following conditions:
  1. for every $(s, req_u, s') \in \delta$, $(s, (req_u, x_u, 1), s') \in \delta'$,
  2. for every $(s, ans_u, s') \in \delta$, $(s, (ans_u, x_u, 0), s') \in \delta'$ and
  3. for every $(s, \tau, s') \in \delta$, $(s, \tau, s') \in \delta'$.

Now, if we could define an encoding of any k-counter automaton into k-client SPS then, in conjunction with the above scheme, we would have a two step encoding of multi-counter automata to SAS. Thus, in two steps, $MCA \Rightarrow SPS \Rightarrow SAS$, given a k-counter automaton $\mathcal{A}$, we can define a k-client SAS M' such that $R_{\mathcal{A}} = R_{M'}$.

4.2.3 Encoding Multi-Counter Automata into SPS

The encoding is pretty straightforward. Given a k-counter automaton $\mathcal{A} = (Q, q_I, \delta)$, we define a k-client SPS with the same state set and $\Gamma_0 = \{u_1, \cdots, u_k\}$. As regards transitions, $\mathbf{inc}_i$ transitions are replaced by req_{u_i} transitions, $\mathbf{dec}_i$ transitions are replaced by ans_{u_i} transitions, and τ-transitions are left as they are.
The equivalent k-client SPS $M = (S, \delta', I)$ is defined as follows:

- $S = Q$,

- $I = \{q_I\}$.
- In order to define δ', we first define a map $h : L \to \Gamma$ as follows:
  - for all $1 \leq j \leq k$, $h(\mathbf{inc}_j) = req_{u_j}$,
  - for all $1 \leq j \leq k$, $h(\mathbf{dec}_j) = ans_{u_j}$, and
  - $h(\tau) = \tau$.

  Now, δ' is the following set: $\delta' = \{(q, h(a), q') \mid (q, a, q') \in \delta\}$.

Theorem 4.2.3. *Given a configuration $(q, \widetilde{n}) \in \Omega_{\mathcal{A}}$, $(q, \widetilde{n})$ is reachable from $(q_I, \widetilde{n}^\dagger)$ in $\mathcal{A}$ if and only if (q, C, χ) is reachable from $(q_I, \emptyset, \emptyset)$ in M, where for every $0 \leq j \leq k$, $\widetilde{n}^\dagger[j] = 0$ and (C, χ) and $\widetilde{n}$ are related as follows: $C \subset_{fin} CN$ and $\forall 1 \leq j \leq k$, $\widetilde{n}[j] = |\{a \in C \mid \chi(a) = u_j\}|$.*

Proof. $\Rightarrow$:) Let $\sigma = (q_I, \widetilde{n}^\dagger) \xRightarrow{\gamma_1} (q_1, \widetilde{n}_1) \xRightarrow{\gamma_2} \cdots \xRightarrow{\gamma_m} (q_m, \widetilde{n}_m) \equiv (q, \widetilde{n})$ be the run reaching $(q, \widetilde{n})$ from initial configuration $(q, \widetilde{n}^\dagger)$ in $\mathcal{A}$. We construct a run ρ, in M, $\rho = (q_I, \emptyset, \emptyset) \xRightarrow{r_1} (q_1, C_1, \chi_2) \xRightarrow{r_2} \cdots \xRightarrow{r_m} (q_m, C_m, \chi_m)$, inductively, as follows: Suppose we have computed $\rho^i = (q_I, \emptyset, \emptyset) \xRightarrow{r_1} (q_1, C_1, \chi_2) \xRightarrow{r_2} \cdots \xRightarrow{r_i} (q_i, C_i, \chi_i)$, for $i < m$. Looking at $(q_i, \widetilde{n}_i) \xRightarrow{\gamma_{i+1}} (q_{i+1}, \widetilde{n}_{i+1})$ the extension $\rho^{i+1} \equiv \rho^i \xRightarrow{r_{i+1}} (q_{i+1}, C_{i+1}, \chi_{i+1})$ is defined as per the following cases:

- $\gamma_{i+1} = \tau$: $r_{i+1} = \tau$, $C_{i+1} = C_i$ and $\chi_{i+1} = \chi_i$.
- $\gamma_{i+1} = \mathbf{inc}_j$: $r_{i+1} = req_{u_j}$. Let a be the first, in order, in $CN - C_i$, then, $C_{i+1} = C_i \cup \{a\}$. $\chi_{i+1} = \chi_i[a \mapsto u_j]$.
- $\gamma_{i+1} = \mathbf{dec}_j$: $r_{i+1} = ans_{u_j}$. Let $X = \{b \in C \mid \chi(b) = u_j\}$, then, $C_{i+1} = C_i - \{a\}$, where a is the first, in order, in X. $\chi_{i+1} = \chi_i \lceil C_{i+1}$. Note that we can consistently remove a from C_i because $\mathbf{dec}_j$ transition in $\mathcal{A}$ makes sure that there is at least one element of the type u_j active in the system.

It is easy to see that ρ is a valid run in M given σ is a valid run in $\mathcal{A}$.

$\Leftarrow$:) Let $\rho = (q_I, \emptyset, \emptyset) \xRightarrow{r_1} (q_1, C_1, \chi_2) \xRightarrow{r_2} \cdots \xRightarrow{r_m} (q_m, C_m, \chi_m)$ be the run, in M, reaching (q, C, χ) from $(q_I, \emptyset, \emptyset)$. We construct a run, in $\mathcal{A}$, $\sigma = (q_I, \widetilde{n}^\dagger) \xRightarrow{\gamma_1} (q_1, \widetilde{n}_1) \xRightarrow{\gamma_2} \cdots \xRightarrow{\gamma_m} (q_m, \widetilde{n}_m) \equiv (q, \widetilde{n})$, inductively, as follows:

Suppose, for all $i < m$, we have computed $\sigma^i = (q_I, \widetilde{n}^\dagger) \xRightarrow{\gamma_1} (q_1, \widetilde{n}_1) \xRightarrow{\gamma_2} \cdots \xRightarrow{\gamma_i} (q_i, \widetilde{n}_i)$. Looking at $(q_i, C_i, \chi_i) \xRightarrow{r_{i+1}} (q_{i+1}, C_{i+1}, \chi_{i+1})$ the extension $\sigma^{i+1} \equiv \sigma^i \xRightarrow{\gamma_{i+1}} (q_{i+1}, \widetilde{n}_{i+1})$ is defined as per the following cases:

- $r_{i+1} = \tau$: $\gamma_{i+1} = \tau$, for all $1 \leq j \leq k$ $\widetilde{n}_{i+1}[j] = \widetilde{n}_i[j]$.

- $r_{i+1} = req_{u_j}$: $\gamma_{i+1} = \mathbf{inc}_j$. $1 \leq j \leq k$ $\widetilde{n}_{i+1}[j] = \widetilde{n}_i[j] + 1$ and for all $j' \neq j$ $\widetilde{n}_{i+1}[j'] = \widetilde{n}_i[j']$.

- $r_{i+1} = ans_{u_j}$: $\gamma_{i+1} = \mathbf{dec}_j$. $1 \leq j \leq k$ $\widetilde{n}_{i+1}[j] = \widetilde{n}_i[j] - 1$ and for all $j' \neq j$ $\widetilde{n}_{i+1}[j'] = \widetilde{n}_i[j']$. Again, note that the $\mathbf{dec}_j$ transition can take place because ans_{u_j} transition occurs in the original run at that point ensures that there is at least one active element of type u_j in the system.

It is easy to see that σ is a valid run in $\mathcal{A}$ given ρ is a valid run in M. □

4.2.4 Encoding SAS into Multi-Counter Automata

One the other hand, given an SAS M with k-client transition systems we can define a k-counter automaton $\mathcal{A}$ such that $R_{\mathcal{A}} = R_M$.

This encoding is a bit involved. Given an SAS M of size n with k-client transition systems of sizes at most m, we can describe a k-counter automaton $\mathcal{A}$ of size of the order $n \cdot m^{n \cdot k}$. The blow up results from the fact that $\mathcal{A}$ has to keep track of states of clients of each type. Fortunately, a SAS machine can remember, at any time, at most $|\Pi_u|$ clients of type u, hence the bound.

Let the k-client SAS be $M = (S, \delta, I)$ with client transition systems $M_u = (Q_u, \delta_u, I_u)$, for each $u \in \Gamma_0$, we give a k-counter automaton $\mathcal{A} = (Q', \delta', q_I)$ as follows:

- Suppose, the variable symbols in each Π_{u_j} are ordered in some way. Let $|\Pi_{u_j}| = k_j$ and $\kappa = \Sigma_{1 \leq j \leq k} k_j$. Then, $Q' = S \times \Pi_{u_j \in \Gamma_0} (Q_{u_j} \cup \{\bot\})^{k_j}$. So, a single state in Q' would look like $\langle s, \widetilde{q}^{u_1}, \cdots \widetilde{q}^{u_k} \rangle$, where, for each $1 \leq j \leq k$ and for each $1 \leq j' \leq k_j$, $\widetilde{q}^{u_j}[j']$ is either a client state of type u_j from Q_{u_j} or $\bot$.

- Using δ and δ_u's we compute δ'. Thereafter, we extend it to configurations $\Omega_{\mathcal{A}}$ and check whether it is consistent with the δ over Ω_M.

Given a transition $s \xrightarrow{(\theta,x,\nu)} s' \in \delta$ with $x \in \Pi_{u_j}$ and $\theta \in \Theta_{u_j}$, for some $1 \leq j \leq k$. Suppose x is the j'th element, in order, in Π_{u_j}:

- if $\nu = 1$ then add transitions $(s, \widetilde{p}^{u_1} \cdots, \widetilde{p}^{u_k}) \xrightarrow{\mathbf{inc}_j} (s', \widetilde{q}^{u_1}, \cdots, \widetilde{q}^{u_k})$ such that there exists $p \in I_{u_j}$ and $p \xrightarrow{\theta} q$, where $q = \widetilde{q}^{u_j}[j']$ and $\widetilde{p}^{u_j}[j'] = \bot$. All other entries in $\widetilde{q}^{u_j}$ and $\widetilde{q}^{u_1} \cdots \widetilde{q}^{u_k}$ remain unchanged over the transition.
- if $\nu = 0$ then
  - ∗ add transitions $(s, \widetilde{p}^{u_1} \cdots, \widetilde{p}^{u_k}) \xrightarrow{\mathbf{dec}_j} (s', \widetilde{q}^{u_1}, \cdots, \widetilde{q}^{u_k})$ such that there exists $q \in F_{u_j}$ and $p \xrightarrow{\theta} q$, where $p = \widetilde{q}^{u_j}[j']$ and $\widetilde{q}^{u_j}[j'] = \bot$. All other entries in $\widetilde{q}^{u_j}$ and $\widetilde{q}^{u_1} \cdots \widetilde{q}^{u_k}$ remain unchanged over the transition.
  - ∗ add transitions $(s, \widetilde{p}^{u_1} \cdots, \widetilde{p}^{u_k}) \xrightarrow{\tau} (s', \widetilde{q}^{u_1}, \cdots, \widetilde{q}^{u_k})$ such that $p \xrightarrow{\theta} q$, where $p = \widetilde{q}^{u_j}[j']$ and $q = \widetilde{q}^{u_j}[j']$. All other entries in $\widetilde{q}^{u_j}$ and $\widetilde{q}^{u_1} \cdots \widetilde{q}^{u_k}$ remain unchanged over the transition.

Theorem 4.2.4. *For every $(s, C, \chi, \pi) \in \Omega_M$, (s, C, χ, π) is reachable in the given SAS M if and only if $(\widetilde{s}, \widetilde{n})$ is reachable in the multi-counter automaton $\mathcal{A}$.*

Proof. $(\Rightarrow)$
Let (s, C, χ, π) be reachable from $(s_0, \emptyset, \emptyset, \emptyset)$, for some $s_0 \in I$, via a run $\rho = (s_0, \emptyset, \emptyset, \emptyset) \xRightarrow{(\theta_1,x_1,\nu_1)} (s_1, C_1, \chi_1, \pi_1) \xRightarrow{(\theta_2,x_2,\nu_2)} \cdots \xRightarrow{(\theta_m,x_m,\nu_m)} (s_m, C_m, \chi_m, \pi_m) \equiv (s, C, \chi, \pi)$. We construct a run, in $\mathcal{A}$, $\sigma = (\widetilde{s}_0, \widetilde{n}_0) \xRightarrow{\gamma_1} (\widetilde{s}_1, \widetilde{n}_1) \xRightarrow{\gamma_2} \cdots \xRightarrow{\gamma_m} (\widetilde{s}_m, \widetilde{n}_m)$ as follows:

- $\widetilde{s}_0 = \langle s_0, \widetilde{q}_0^{u_1}, \cdots, \widetilde{q}_0^{u_k} \rangle$ where for all $1 \leq j \leq k$, $\widetilde{q}_0^{u_j} = \bot^{k_j}$. Also, for all $1 \leq j \leq k$, $\widetilde{n}_0[j] = 0$.
- Suppose we have defined, inductively, $\sigma^i = (\widetilde{s}_0, \widetilde{n}_0) \xRightarrow{\gamma_1} (\widetilde{s}_1, \widetilde{n}_1) \xRightarrow{\gamma_2} \cdots \xRightarrow{\gamma_i} (\widetilde{s}_i, \widetilde{n}_i)$, $i < m$. We have to extend it to σ^{i+1}.

  Look at $(s_i, C_i, \chi_i, \pi_i) \xRightarrow{(\theta_{i+1},x_{i+1},\nu_{i+1})} (s_{i+1}, C_{i+1}, \chi_{i+1}, \pi_{i+1})$. Let $x_{i+1} \in \Pi_{u_j}$ and x_{i+1} be the j'th element in order.

  - $\nu_{i+1} = 1$:) In this case, there exists an $a \in CN$ such that $a \in C_{i+1} - C_i$ and there exists $p \in I_{u_j}$ such that $p \xrightarrow{\theta_{i+1}} \chi_{i+1}(a)$. Also $\pi_{i+1}(x_{i+1}) = a$. $\sigma^{i+1} \overset{\text{def}}{=} \sigma^i \xRightarrow{\mathbf{inc}_j} (\widetilde{s}_{i+1}, \widetilde{n}_{i+1})$ where

- - ∗ $\widetilde{n}_{i+1}[j] = \widetilde{n}_i[j] + 1$, for all $j' \neq j$, $\widetilde{n}_{i+1}[j'] = \widetilde{n}_i[j']$ and
    - ∗ $\widetilde{s}_{i+1} = (s_{i+1}, \widetilde{q}^{u_1}_{i+1}, \cdots, \widetilde{q}^{u_k}_{i+1})$ $\widetilde{q}^{u_j}_{i+1}[j'] = \chi_{i+1}(a)$ and all other entries in $\widetilde{q}^{u_j}_{i+1}$ and $\widetilde{q}^{u_1}_{i+1} \cdots \widetilde{q}^{u_k}_{i+1}$ remain unchanged over the transition.
  - $\nu_{i+1} = 0$:) Let $a = \pi_i(x_{i+1}) = \pi_{i+1}(x_{i+1})$. There are two possibilities:
    - ∗ $a \in C_i \cap C_{i+1}$:) $\sigma^{i+1} \overset{\text{def}}{=} \sigma^i \overset{\tau}{\Longrightarrow} (\widetilde{s}_{i+1}, \widetilde{n}_{i+1})$ where
      - · $\widetilde{n}_{i+1}[j] = \widetilde{n}_i[j]$, for all $j' \neq j$, $\widetilde{n}_{i+1}[j'] = \widetilde{n}_i[j']$ and
      - · $\widetilde{s}_{i+1} = (s_{i+1}, \widetilde{q}^{u_1}_{i+1}, \cdots, \widetilde{q}^{u_k}_{i+1})$ $\widetilde{q}^{u_j}_{i+1}[j'] = \chi_{i+1}(a)$ and all other entries in $\widetilde{q}^{u_j}_{i+1}$ and $\widetilde{q}^{u_1}_{i+1} \cdots \widetilde{q}^{u_k}_{i+1}$ remain unchanged over the transition.
    - ∗ $a \in C_i - C_{i+1}$:) $\sigma^{i+1} \overset{\text{def}}{=} \sigma^i \overset{\mathbf{dec}_j}{\Longrightarrow} (\widetilde{s}_{i+1}, \widetilde{n}_{i+1})$ where
      - · $\widetilde{n}_{i+1}[j] = \widetilde{n}_i[j] - 1$, for all $j' \neq j$, $\widetilde{n}_{i+1}[j'] = \widetilde{n}_i[j']$ and
      - · $\widetilde{s}_{i+1} = (s_{i+1}, \widetilde{q}^{u_1}_{i+1}, \cdots, \widetilde{q}^{u_k}_{i+1})$ $\widetilde{q}^{u_j}_{i+1}[j'] = \bot$ and all other entries in $\widetilde{q}^{u_j}_{i+1}$ and $\widetilde{q}^{u_1}_{i+1} \cdots \widetilde{q}^{u_k}_{i+1}$ remain unchanged over the transition.

Now, it is easy to see that σ is a valid run in $\mathcal{A}$ given ρ is a valid run in M.

$(\Leftarrow)$

Let $(\widetilde{s}, \widetilde{n})$ be reachable from $(\widetilde{s}_0, \widetilde{n}_0)$, for some $\widetilde{s}_0 \in I'$, via a run $\sigma = (\widetilde{s}_0, \widetilde{n}_0) \overset{\gamma_1}{\Longrightarrow} (\widetilde{s}_1, \widetilde{n}_1) \overset{\gamma_2}{\Longrightarrow} \cdots \overset{\gamma_m}{\Longrightarrow} (\widetilde{s}_m, \widetilde{n}_m) \equiv (\widetilde{s}, \widetilde{n})$. We construct a run ρ, in M, $\rho = (s_0, \emptyset, \emptyset, \emptyset) \overset{(\theta_1, x_1, \nu_1)}{\Longrightarrow} (s_1, C_1, \chi_1, \pi_1) \overset{(\theta_2, x_2, \nu_2)}{\Longrightarrow} \cdots \overset{(\theta_m, x_m, \nu_m)}{\Longrightarrow} (s_m, C_m, \chi_m, \pi_m)$ as follows:

- s_0 is the first element of the tuple $\widetilde{s}_0$.
- Suppose, we have defined $\rho^i = (s_0, \emptyset, \emptyset, \emptyset) \overset{(\theta_1, x_1, \nu_1)}{\Longrightarrow} (s_1, C_1, \chi_1, \pi_1) \overset{(\theta_2, x_2, \nu_2)}{\Longrightarrow} \cdots \overset{(\theta_i, x_i, \nu_i)}{\Longrightarrow} (s_i, C_i, \chi_i, \pi_i)$, $i < m$. Now, we need to extend it to $i + 1$.

  Look at $(\widetilde{s}_i, \widetilde{n}_i) \overset{\gamma_{i+1}}{\Longrightarrow} (\widetilde{s}_{i+1}, \widetilde{n}_{i+1})$. s_{i+1} would be the first element of $\widetilde{s}_{i+1}$.

  Fix $\rho^{i+1} \overset{\text{def}}{=} \rho^i \overset{(\theta_{i+1}, x_{i+1}, \nu_{i+1})}{\Longrightarrow} (s_{i+1}, C_{i+1}, \chi_{i+1}, \pi_{i+1})$ according to following cases:

  - $\gamma_{i+1} \equiv \tau$:) Clearly, in this case $\widetilde{n}_i \equiv \widetilde{n}_{i+1}$. Hence, $\nu_{i+1} = 0$. $C_{i+1} = C_i$. By the definition of δ', there exists at most one $1 \leq j \leq k$ and exactly one $1 \leq j' \leq k_j$ such that $\widetilde{q}^{u_j}_i[j'] = p$ differs from $\widetilde{q}^{u_j}_{i+1}[j'] = p'$. Also, there exists $\theta \in \Theta_{u_j}$ such that $(p, \theta, p') \in \delta$. Define $\theta_{i+1} = \theta$. Define $\chi_{i+1} = \chi_i[\pi_i(x^{u_j}_{j'}) \mapsto p']$ and $\pi_{i+1} \equiv \pi_i$.

- $\gamma_{i+1} \equiv \mathbf{dec}_j$:) In this case, $\widetilde{n}_i[j] - 1 = \widetilde{n}_{i+1}[j]$, and for all $j' \neq j$, $\widetilde{n}_i[j'] = \widetilde{n}_{i+1}[j']$. Here, again, $\nu_{i+1} = 0$. By the definition of δ', there exists at most one $1 \leq j \leq k$ and exactly one $1 \leq j' \leq k_j$ such that $\widetilde{q}_i^{u_j}[j'] = p$ differs from $\widetilde{q}_{i+1}^{u_j}[j'] = \perp$. Also, there exists $\theta \in \Theta_{u_j}$ and $p' \in F_{u_j}$ such that $(p, \theta, p') \in \delta$. Define $C_{i+1} = C_i - \pi_i(x_{j'}^{u_j})$, $\chi_{i+1} = \chi_i \lceil C_{i+1}$ and $\pi_{i+1} \equiv \pi_i$.

- $\gamma_{i+1} \equiv \mathbf{inc}_j$:) In this case, $\widetilde{n}_i[j] + 1 = \widetilde{n}_{i+1}[j]$, and for all $j' \neq j$, $\widetilde{n}_i[j'] = \widetilde{n}_{i+1}[j']$. Here, $\nu_{i+1} = 1$. By the definition of δ', there exists at most one $1 \leq j \leq k$ and exactly one $1 \leq j' \leq k_j$ such that $\widetilde{q}_i^{u_j}[j'] = \perp$ differs from $\widetilde{q}_{i+1}^{u_j}[j'] = p'$. Also, there exists $\theta \in \Theta_{u_j}$ and $p \in I_{u_j}$ such that $(p, \theta, p') \in \delta$. Define $C_{i+1} = C_i \cup \{a\}$, where a is the first in order in $CN - C_i$. Now, let $\chi_{i+1} = \chi_i[a \mapsto p']$ and $\pi_{i+1} \equiv \pi_i[x_{j'}^{u_j} \mapsto a]$.

Note that $x_{j'}^{u_j}$ is the j'th variable symbol, in order, in Π_{u_j}. Now, it is easy to see that ρ is a valid run in M only if σ is a valid run in $\mathcal{A}$.

□

Thus, we conclude that k-client SAS are equivalent to (or have the same behaviour as) k-counter automata. Therefore, the class of SAS machines have the same closure properties as class of counter machines with no zero test. In particular, we can assert the following:

Corollary 4.2.5. *1. The class of SAS systems is closed under union as well as intersection,*

2. The class of SAS systems is not closed under complementation, and

3. The reachability problem is decidable for SAS systems.

We can also look at the bounded cases of SPS/SAS. As already mentioned in the beginning, while verifying SPS/SAS against a safety/liveness property, we expect the formula to provide us with a bound over the verified system. Therefore, we do the necessary reachability analysis of this bounded system in place of the original infinite-state system.

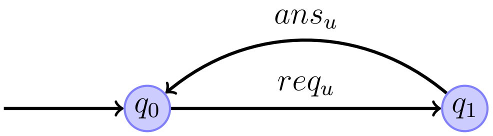


Figure 4.2: SPS with $|\Gamma_0| = 1$ and "one" pending request

4.3 Modelling Examples for Discrete Services

In this section we describe many example instances of discrete services modelled using SPS. To recall, SPS comprises a single server agent communicating with unbounded (finite but unknown) number of passive clients of many "types". The communication is extremely rudimentary, consisting of a "request" and an "answer". This is accomplished by defining an automaton with transition relation labelled with "requests" and "answers" of different client types. Furthermore, the identities of the clients are remembered by storing them "implicitly" in a queue.

4.3.1 Generic Modelling Examples using SPS

We begin by describing a series of SPS machines modelling a generic case with a finite set of types Γ_0 and the corresponding alphabet Γ. We begin with Γ_0 size one and look at transition systems which remember a small number of pending requests. Then, we move on to Γ_0 of size two and do the same. We find that the size of transition system clearly depends on the size of Γ_0 and the number of requests that are pending at any instance. As we move on to higher Γ_0 sizes the system size blows up and providing little insight into the working of the machine.

Models With $|\Gamma_0| = 1$

Let $\Gamma_0 = \{u\}$, say, and the corresponding Γ as $\{req_u, ans_u\}$. We first give a representative SPS transition system where the machine remembers a single pending request. Consequently, the SPS has two states q_0 and q_1. q_1 is the state with pending request and is distinguished from the state q_0, where there are no pending requests.

Thereafter, we give transition systems which remember at most two pending requests. The states q_0 and q_1 have the same meaning as before. Of the two new

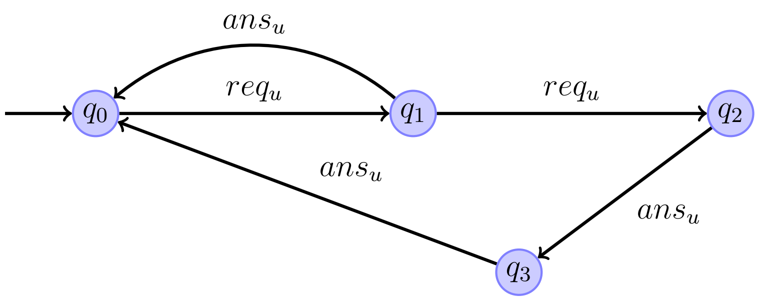


Figure 4.3: SPS with $|\Gamma_0| = 1$ and at most “two” pending requests

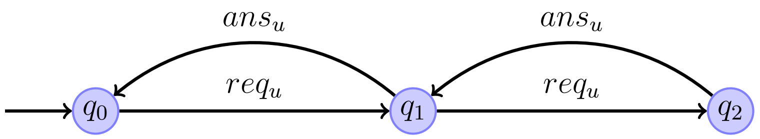


Figure 4.4: SPS with $|\Gamma_0| = 1$ and “two” pending requests

states, q_1 remembers two pending requests whereas q_3 remembers one, but differs from q_1 in the sense that q_1 expects a further request whereas q_3 does not.

In Figure 4.3, there will never be more than two pending requests at any point of time. We can give an alternate transition system, in Figure 4.4, where there can be more pending requests. In the same vein we can modify the transition system further and give examples where the machine remembers ≥ 3 pending requests.

In either case, we notice that we can always trace back to the initial state q_0, where there are no pending requests.

Why do we need to have transition systems with runs so as every request is balanced by an answer? Let us describe a few of those which have no matching request-answers.

Let us start with an SPS with a state-set size 1, which only takes up requests, piling them in turn. The machine is given in the Figure 4.5. In this case, the state q_0 refers to the situation where the system has one or more pending requests. This SPS will admit runs of the following kind:
$(q_0, \emptyset, \emptyset) \overset{req_u}{\Longrightarrow} (q_0, \{1\}, \{1 \mapsto u\}) \overset{req_u}{\Longrightarrow} (q_0, \{1, 2\}, \{1, 2 \mapsto u\}) \overset{req_u}{\Longrightarrow} (q_0, \{1, 2, 3\}, \{1, 2, 3 \mapsto u\}) \overset{req_u}{\Longrightarrow} (q_0, \{1, 2, 3, 4\}, \{1, 2, 3, 4 \mapsto u\}) \cdots$.

Now, we give a similar transition system, but with two states, q_0 and q_1. q_1 is

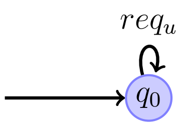


Figure 4.5: SPS with $|\Gamma_0| = 1$ and at least “one” pending request

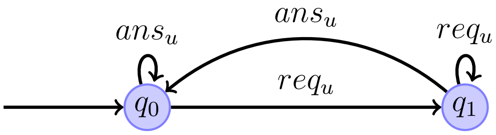


Figure 4.6: SPS with $|\Gamma_0| = 1$ and at least “one” pending request

the state in which system has one or more pending requests and q_0 is the state in which the system has answered one or more requests. The machine is given in the Figure 4.6. This SPS will admit runs of the following kind:

- $(q_0,\emptyset,\emptyset)\xRightarrow{req_u}(q_1,\{1\},\{1\mapsto u\})\xRightarrow{ans_u}(q_0,\emptyset,\emptyset)\xRightarrow{req_u}(q_1,\{1\},\{1\mapsto u\})\xRightarrow{ans_u}(q_0,\emptyset,\emptyset)\cdots$
  $(q_0,\emptyset,\emptyset)\xRightarrow{req_u}(q_1,\{1\},\{1\mapsto u\})\xRightarrow{req_u}(q_1,\{1,2\},\{1,2\mapsto u\})\xRightarrow{ans_u}(q_0,\{2\},\{2\mapsto u\})\xRightarrow{ans_u}(q_0,\emptyset,\emptyset)\cdots$
  Note, that the aforementioned runs have matching answers for all requests. The following runs don't necessarily have matching answers for all requests.

- $(q_0,\emptyset,\emptyset)\xRightarrow{req_u}(q_1,\{1\},\{1\mapsto u\})\xRightarrow{req_u}(q_1,\{1,2\},\{1,2\mapsto u\})\xRightarrow{ans_u}(q_0,\{2\},\{2\mapsto u\})\xRightarrow{req_u}(q_1,\{1,2\},\{1,2\mapsto u\})\xRightarrow{req_u}(q_1,\{1,2,3\},\{1,2,3\mapsto u\})\xRightarrow{ans_u}(q_0,\{2,3\},\{2,3\mapsto u\})\xRightarrow{req_u}(q_1,\{1,2,3\},\{1,2,3\mapsto u\})\cdots$
  $(q_0,\emptyset,\emptyset)\xRightarrow{req_u}(q_1,\{1\},\{1\mapsto u\})\xRightarrow{req_u}(q_1,\{1,2\},\{1,2\mapsto u\})\xRightarrow{req_u}(q_1,\{1,2,3\},\{1,2,3\mapsto u\})\xRightarrow{req_u}(q_1,\{1,2,3\},\{1,2,3\mapsto u\})\cdots$

Models With $|\Gamma_0| = 2$

In this section, we present SPS instances of those generic cases where there are client requests of two different types. Let $\Gamma_0 = \{u, v\}$ and the attendant Γ be defined in the usual manner. We first present a system, in Figure 4.7 which remembers a single request first from u then v. We can have a copy of this system with requests from u and v exchanging places. This machine is given in the Figure 4.8. We can join these two machines and give a more comprehensive

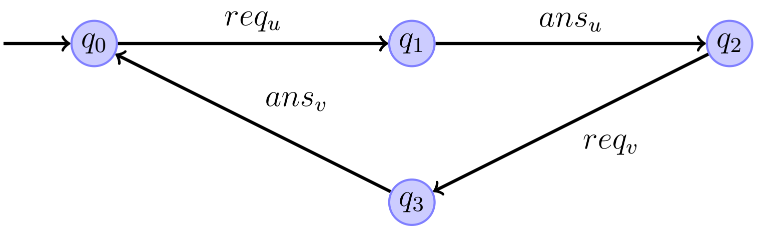


Figure 4.7: An SPS with $|\Gamma_0| = 2$

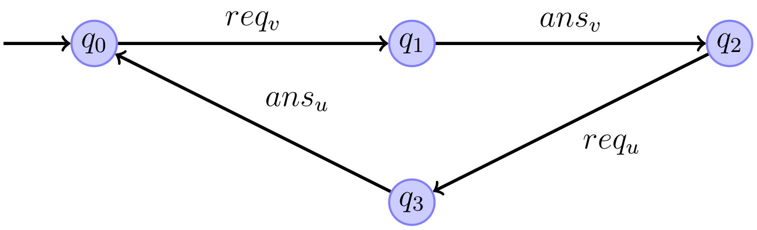


Figure 4.8: Another SPS with $|\Gamma_0| = 2$

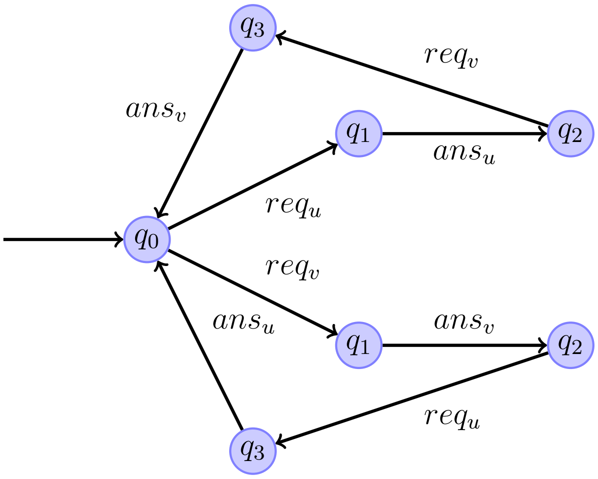


Figure 4.9: A comprehensive SPS with $|\Gamma_0| = 2$

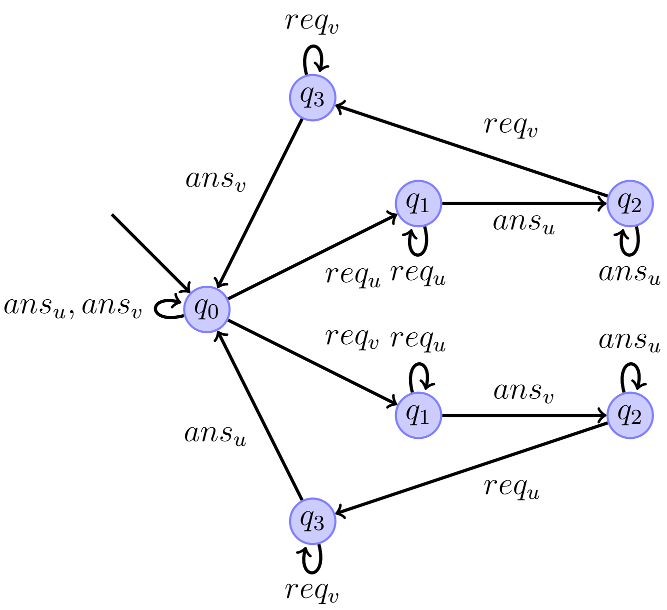


Figure 4.10: An SPS with $|\Gamma_0| = 2$ and unmatched requests-answers

system definition as in Figure 4.9.

Notice that the transition system admits only those runs which have matching answers for each request of the two given types $\{u, v\}$. We can add self-loops to the states of the systems, with appropriate labellings, so as to generate runs which lack matching answers for requests. The modified transition system is given in Figure 4.10.

Similarly, we can give transition systems for machines which have at most two pending requests at any point of time, as in Figures 4.11 and 4.12. Then, we can modify the two transition systems by incorporating self loops to allow for non-matching request-answer scenarios. These machines are given in Figures 4.13 and 4.14.

4.3.2 Loan Approval Service

After we have seen some generic SPS discrete client-server systems, we proceed to describe a special case. We give a series of SPS for automated Loan Approval Web Service System. In this composite system, there is a designated Web server acting as Loan Officer which admits loan requests of various sizes, say h depicting high

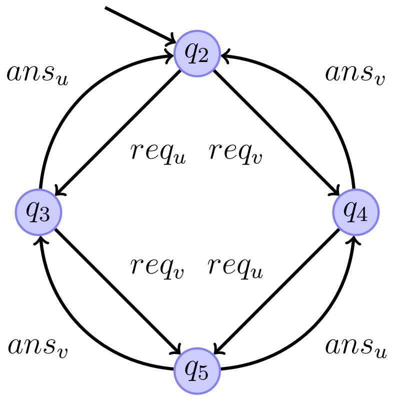


Figure 4.11: An SPS with $|\Gamma_0| = 2$ and at most two pending requests

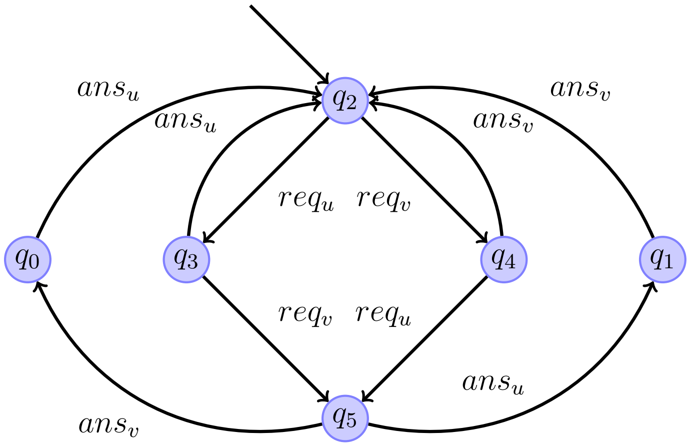


Figure 4.12: Another SPS with $|\Gamma_0| = 2$ and at most two pending requests

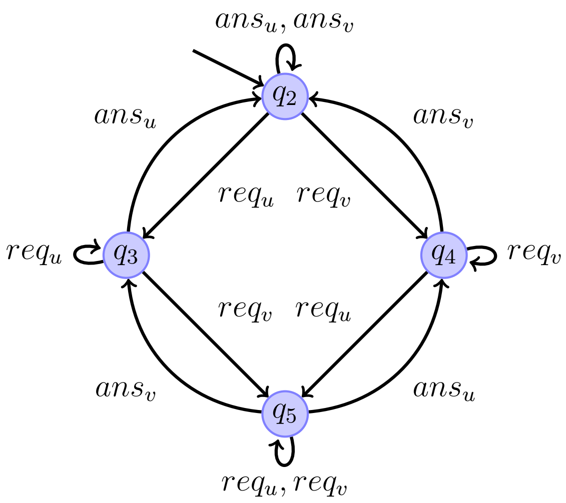


Figure 4.13: Modified SPS with $|\Gamma_0| = 2$ and non-matching request-answers

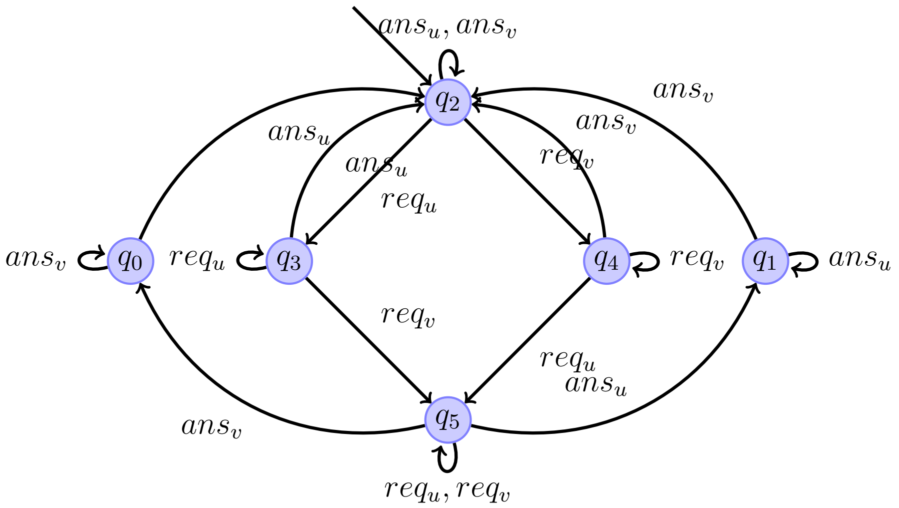


Figure 4.14: Another modified SPS with $|\Gamma_0| = 2$ and non-matching requests-answers

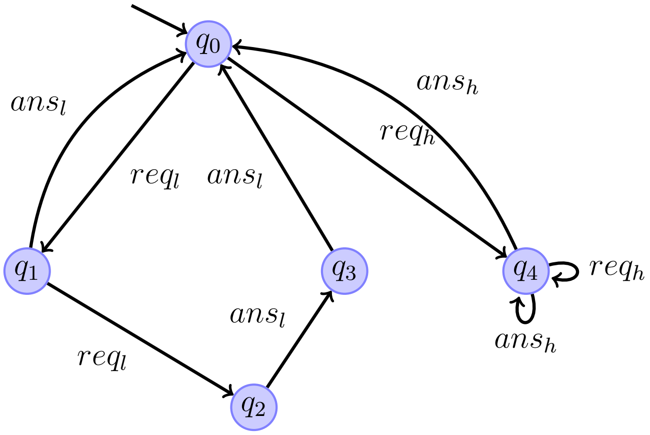


Figure 4.15: An SPS modelling Loan Approval System

(large) amount and l depicting low (small) amounts. Depending on the number of loan requests (high and low) and according to an a priori fixed loan disbursal policy, the loan officer accepts or rejects the pending requests. The behaviour of the loan officer is captured as SPS.

Let $\Gamma_0 = \{h, l\}$, where h denotes high end loan and l denotes low size loan, and the corresponding alphabet $\Gamma = \{req_h, req_l, ans_h, ans_l\}$, the Loan Approval System can be modelled as an SPS $M_1 = (Q_1, \delta_1, \{q_0\})$ as shown in the Figure 4.15.

Here, we briefly describe the working of the automaton M_1. M_1, starting from q_0, keeps track of at most two low-amount requests. q_1 is the state with one pending request whereas q_4 is the state with two pending requests. Whenever the system gets a high amount request, it seeks to dispose it at the earliest and tries avoiding to take up a low request as long as a high one is pending with it. But, it may not succeed all the time, i.e, when the automaton reaches q_6, it is possible that it can loop back to initial state q_0, with one or more *high* pending requests, and then take up *low* requests.

It is not difficult to see that there are runs of M_1 which satisfy the following property, ψ_1, and there are those which don't. ψ_1 asserts that "whenever there is a request of type *low* there is an answer of type *low* in the next instant".

Note that the following path in M_1 would give rise to a run of M_1 where ψ_1

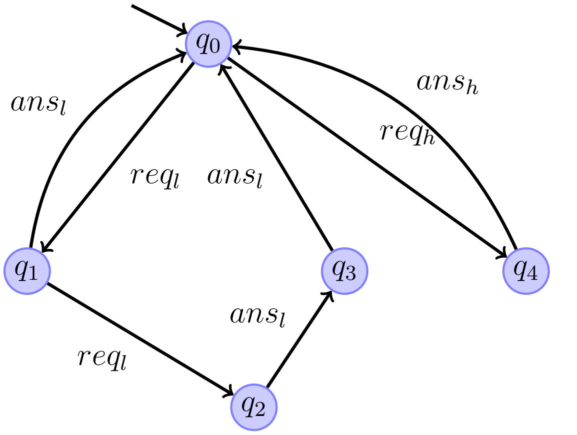


Figure 4.16: A modified SPS modelling Loan Approval System

holds;

$$\sigma_1 = q_0 \xrightarrow{req_h} q_4 \xrightarrow{req_h} q_4 \xrightarrow{ans_h} q_0 \xrightarrow{req_l} q_1 \xrightarrow{ans_l} q_0 \xrightarrow{req_l} q_1 \xrightarrow{ans_l} q_0$$

Also, the following path in M_1 induces a run of M_1 where ψ_1 does not hold;

$$\sigma_2 = q_0 \xrightarrow{req_h} q_4 \xrightarrow{req_h} q_4 \xrightarrow{ans_h} q_0 \xrightarrow{req_l} q_1 \xrightarrow{req_l} q_2 \xrightarrow{ans_l} q_3 \xrightarrow{ans_l} q_0$$

Now, suppose there is another property ψ_2 described as "there is no request of type *low* taken up as long as there is a *high* request pending". If we want to avoid ψ_2 in the Loan Approval System then we need to modify M_1 and define $M_2 = (S_2, \delta_2, \{q_0\})$ as in the Figure 4.16. Note that the following path in M_2 induces a run of M_2 where ψ_1 as well as ψ_2 hold;

$$\sigma_3 = q_0 \xrightarrow{req_h} q_4 \xrightarrow{ans_h} q_0 \xrightarrow{req_l} q_1 \xrightarrow{ans_l} q_0 \xrightarrow{req_l} q_1 \xrightarrow{ans_l} q_0$$

Also, the following path in M_2 induces a run of M_2 where ψ_1 does not hold but ψ_2 does hold;

$$\sigma_4 = q_0 \xrightarrow{req_h} q_6 \xrightarrow{ans_h} q_0 \xrightarrow{req_l} q_1 \xrightarrow{req_l} q_2 \xrightarrow{ans_l} q_3 \xrightarrow{ans_l} q_0$$

Clearly if we want to make sure that there are no two *low* requests pending at any time, *i.e.*, our model satisfies ψ_1 as well as ψ_2, then we modify M_2 and describe M_3 as in Figure 4.17.

We shall see later that these properties can be described easily in a decidable

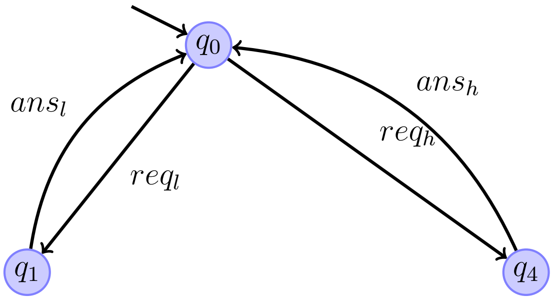


Figure 4.17: Another modified SPS modelling Loan Approval System

logic which we call $\mathcal{L}_{SPS}$.

The reader would observe that the SPS machine has only two actions req_u and ans_u corresponding to each client type $u \in \Gamma_0$. The Loan Approval example may suggest the division of action ans_u into two sub-actions, say, yes_u and no_u, for the two possibilities where loan request of a particular type meets with either an answer "yes" or an answer "no". Thus, with $\Gamma = \{req_u, yes_u, no_u \mid u \in \Gamma_0\} \cup \{\tau\}$, we can describe an SPS with almost the same working, but an enriched model.

Notice that, in SPS, the customer (user or client) simply sends a request of some particular type and waits for an answer. What happens when the client executes some non-trivial actions and communicates with the server in the meanwhile? In the next section we present an example of such client-server systems.

4.4 Modelling Examples for Session-Oriented Services

We propose to capture session-oriented services by the SAS models. In such client-server systems, a client and the server communicate with each other during a session, *i.e.*, between the send-request and receive-answer. This is made possible by the presence of activity alphabet Θ and client transition systems M_u for each client type $u \in \Gamma_0$ in the SAS. Θ is essentially the communication alphabet and M_u captures the communication pattern between the server and the client of type u. Note, also, that with a transition of the kind $\overset{(\theta,x,1)}{\Rightarrow}$, we can add new clients to the system too. Thus, we see that SAS models all requirements of session-oriented

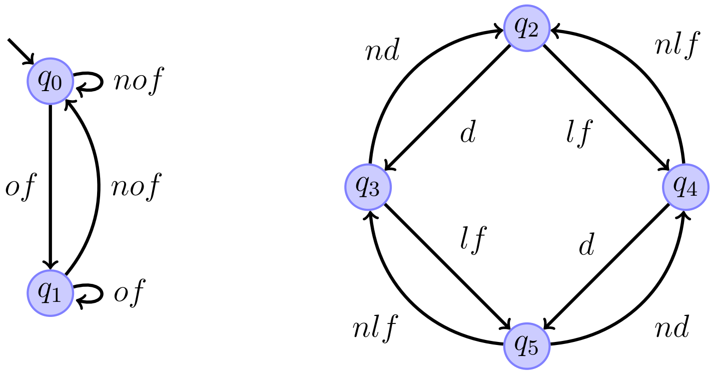


Figure 4.18: Client Transition systems for Hotel (h) and Airline (a)

systems. The following example using a Travel Agency Service will make our claim clearer.

4.4.1 Travel Agency Service

The Travel Agency System consists of a Travel Agency Service ta and two types of Clients , Hotel Accommodation, h, and Airline Reservation, a. The clients of type h look after accommodation needs in hotel/s, whereas those of type a offer bookings on airlines. There are unboundedly many agents of each type competing to cater to the needs of the Travel Agent. The travel agent ta, in turn, has to come up with holiday packages, suitable to the needs and pockets of it's targeted customer base. This, it does by interacting with the competing h and a services.

First, we describe client transition systems M_h and M_a. The system $M_h = (Q_h, \delta_h, I_h)$ is a finite automaton for Hotel Accommodation Agent with the interaction alphabet $\Theta_h = \{nof, of\}$. The activity of means the service introduces discounted off-season rates, whereas nof is the withdrawal of those rates. The machine is initially in a state q_0 where it offers no off-season rates. When the system decides to offer off-season discounts then it executes the action of and moves to state q_1 else it remains in q_0. It can go back to the default state by withdrawing the discount by executing the action nof.

The system $M_a = (Q_a, \delta_a, I_a)$ is the corresponding transition system for Airline Service, with the interaction alphabet $\Theta_a = \{lf, nlf, df, ndf\}$. Here lf means introduction of low fares, d means introduction of direct flights, whereas nlf means

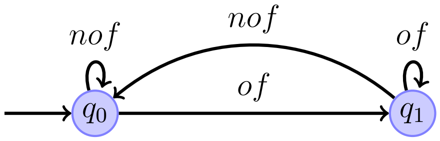


Figure 4.19: Hotel Transition System

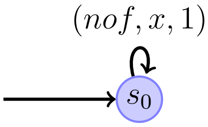


Figure 4.20: A rudimentary SAS for Travel Agency

withdrawal of low fares and nd means cancellation of direct flights. The machine, initially does not offer any low fare direct flights and is in state q_2. The other states are q_3, q_4 and q_5. It can move to different states according to its corresponding activity as is amply clear from the Figure 4.18.

We take the definition of M_h with two states and $\Theta_h = \{of, nof\}$, as given in the Figure 4.19 and describe a transition system for travel agent which remembers at most one agent *i.e.*, $\Pi_h = \{x\}$. The single state SAS in Figure 4.20 admits arbitrary number of clients offering no off-season rates and does nothing else. The SAS not only ignores any change in the state of clients of hotel type, already at hand, but, also, clients of other types providing other services. Note that any run of this machine will have a growing configuration sets though every active client $a \in C$ remains frozen in the identical state q_0.

We can give a richer SAS with more states which keeps track of the behaviour of at most one client of hotel type by changing its own state and accordingly coming up with a rudimentary package for customer. Even this SAS ignores the behaviour of clients of other type as the resultant machine could become unwieldy. The states of SAS in Figure 4.21 have the following meaning where s_0 is the initial state as well as final state.

- s_0: no active agents.
- s_1: exactly one active agent offering OF and no agent offering NOF.
- s_2: exactly one agent offering NOF and no agent offering OF.

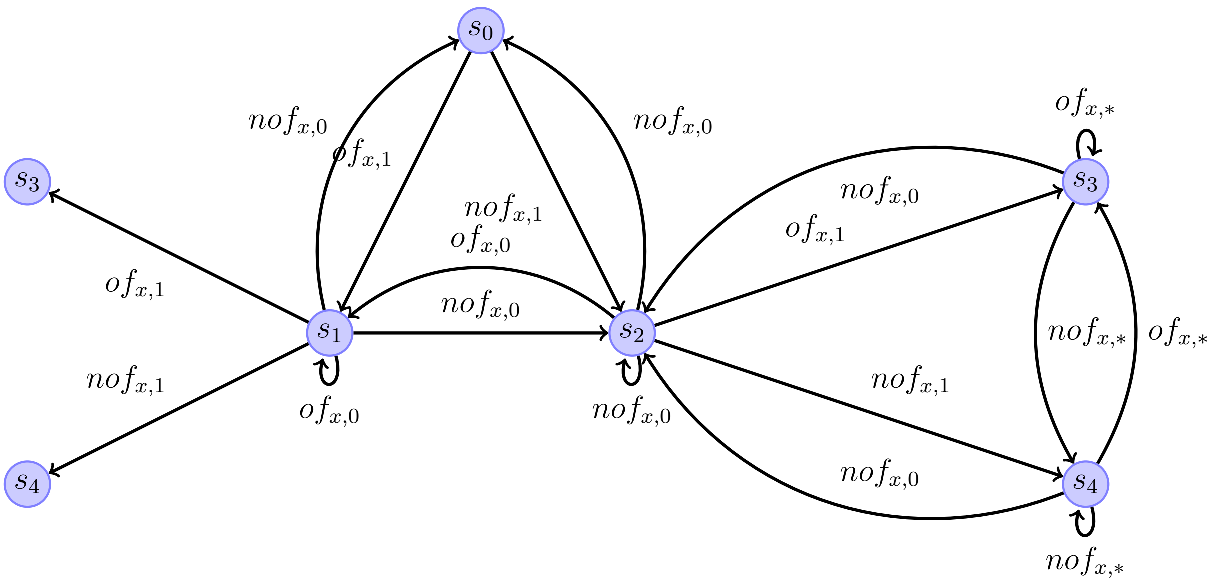


Figure 4.21: An SAS for Travel Agency with at most one client at a time

- s_3: ≥ 1 agents offering OF and ≥ 0 agents offering NOF.
- s_4: ≥ 1 agents offering NOF and ≥ 0 agents offering OF.

With the same M_h as given in Figure 4.19 but with $\Pi_h = \{x_1, x_2\}$ we can define an enhanced prototypical SAS. In this case the server automaton may contain twenty one (21) states.

Note that the above server automata do not have any transition involving activity alphabet Θ_a. That is, the Travel Agent (ta) ignores the temporal behaviour of Airline (a) and takes decisions influenced only by the activity of Hotel Service.

Now, we can consider transition systems for travel agent with two client types h and a and their corresponding activity alphabets $\Theta_h = \{of, nof\}$ and $\Theta_a = \{lf, nlf\}$ and their corresponding abstract name alphabet $\Pi_h = \{x\}$ and $\Pi_a = \{y\}$. The client transition systems are given in the Figure 4.22. It turns out that enhanced SAS has 20 states.

Similarly we can define SASs where the client transition systems have richer structures. We can have client transition systems M_h, M_a and M_t with the extended activity alphabet Θ_h, Θ_a and Θ_t and the corresponding composite transition system M for the travel agent ta.

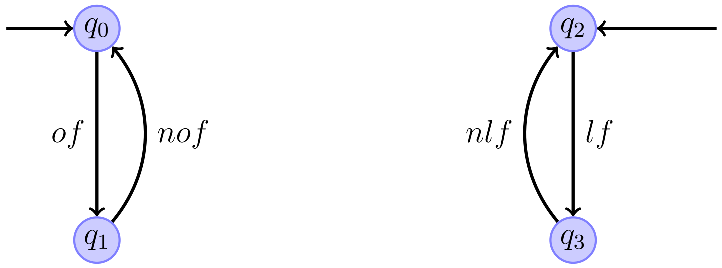


Figure 4.22: Client Transition Systems for Hotel and Airline

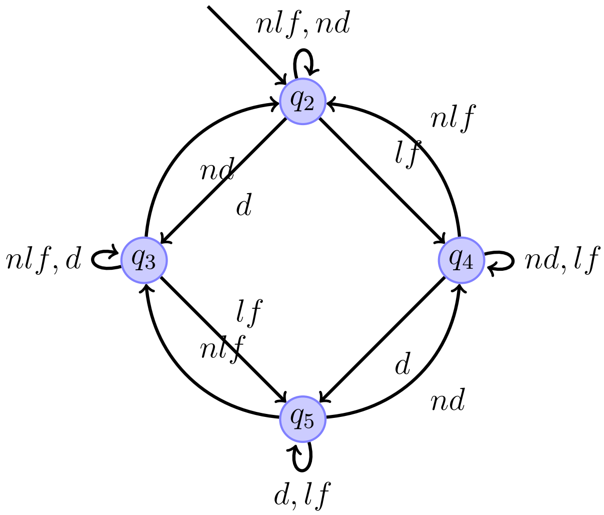


Figure 4.23: Airline Transition System

We can give an SAS with 9 states for the case where we have an M_a with $|\Pi_a| = 1$ and $\Theta_a = \{lf, d, nlf, nd\}$. The client transition system is given in the Figure 4.23. On the same lines, we can give an SAS with 35 states for the case where we have two clients in the composite system, M_h with $\Pi_h = \{x\}$ and $\Theta_h = \{of, nof\}$ and M_a with $\Pi_a = \{y\}$ and $\Theta_a = \{lf, d, nlf, nd\}$. The client transition systems are given in the Figure 4.24. Clearly, describing SAS with even rudimentary client transition systems and more than one abstract client names is a difficult task. It has a large state space and unwieldy transition system. What could be an alternative way of SAS description? We suggest the use of a fragment of Monadic First Order Temporal Logic $MFOTL$ as descriptive language for SAS. This language is quite expressive and has a decidable fragment too, namely the

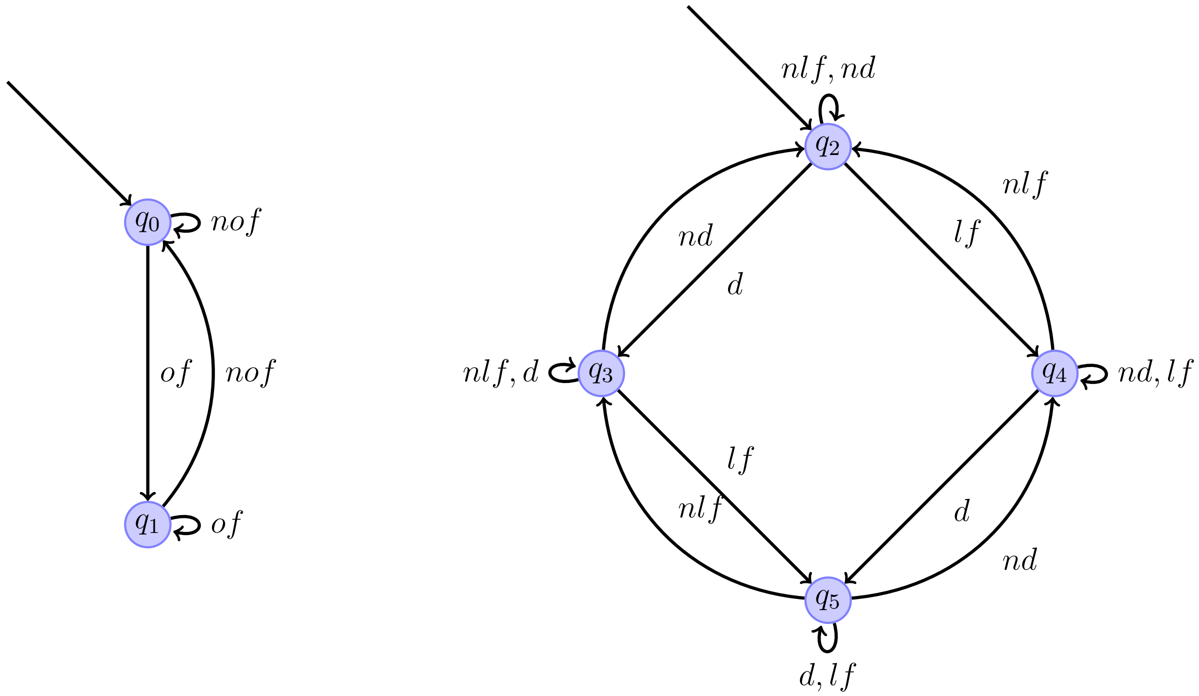


Figure 4.24: Client Transition Systems for Hotel and Airline

monodic fragment. We study $MFOTL$ in later chapters.

In this chapter we presented two automaton models for client-server systems with unbounded number of agents. The first one captures systems with passive clients and the second models those with active clients. These systems have infinite state space in general and their reachability properties are hard to decide, even though basic automaton properties like union and intersection trivially hold. To reason about the reachability and other closure properties, we showed, via a back and forth encoding that our class of automata models are equivalent to multi-counter automata without zero test. We also saw that bounded cases of these systems reduce to simple Büchi automata which could further be used to model check relevant systems. Also, we described two real life system examples, Loan Approval Web service and Travel Agency Web service, and used SPS and SAS respectively, to model them. In the later chapters we shall present specification languages for these systems.

5

Temporal Logics for Systems with Unbounded Agents: Undecidability

Propositional temporal logics have been extensively used for specifying safety and liveness requirements of reactive systems. Backed by a set of tools with theorem proving and model checking capabilities, temporal logic is a natural candidate for specifying service policies. In the context of distributed systems, they have been extended with mechanisms for specifying message exchange between agents. There are several candidate temporal logics for message passing systems, but these work with a priori fixed number of agents, and for any message, the identity of the sender and the receiver are fixed at design time. In order to write specifications for client server systems with unbounded agents, we need to extend such logics with means for referring to agents in some more abstract manner (than by name).

A natural and direct approach to refer to unknown clients is to use logical variables: rather than work with atomic propositions p, we use monadic predicates $p(x)$ to refer to property p being true of client x. We can quantify over such x existentially and universally to specify policies relating clients. We are thus naturally lead to the realm of Monadic First Order Temporal Logics ($MFOTL$)[35]. In fact, it is easily seen that $MFOTL$ [34] is expressive enough to frame almost every requirement specification of client-server systems of the kind discussed above. Unfortunately, $MFOTL$ is undecidable [45], [70], and we need to limit the expressiveness so that we have decidable verification problem.

In this chapter we present $MFOTL$ and show, through examples, its suitability as specification language of unbounded agent client server systems. We also prove this logic to be undecidable by proving the same for a small fragment. Undecidability is proved by encoding the Minsky Machines[69].

5.1 The Logic $MFOTL$

We first describe the syntax and semantics of Monadic First Order Logic ($MFOTL$) with equality (=). Let $Prop$ be a countable set of monadic predicates and Var be another countable set of variable symbols. Let p, q, with or without subscripts etc. be the elements of $Prop$ and x, y, with or without subscripts, be the elements from Var. The set of all well formed formulas of this fragment is defined as follows:

$$\Phi ::= p(x) \mid x = y \mid \neg\alpha \mid \alpha \vee \beta \mid \bigcirc\alpha \mid \alpha\mathbf{U}\beta \mid (\exists x)\alpha$$

The formulas of this logical fragment are interpreted over sequences of MFO models over fixed universe D with a valuation $\pi : Var \to D$ giving meaning to free variables at a particular time instance. Formally, a model is a pair $M = (D, I)$ where D is a non-empty domain and $I = I_0 I_1 I_2 \cdots$ is a sequence of interpretations, where for all $i \geq 0$, $I_i : Prop \to 2^D$ gives the meaning of $p \in Prop$ at the ith instance. I_i can be alternately expressed as $I_i : Prop \times D \to \{\top, \bot\}$. The satisfiability relation $\models$ is defined inductively as follows:

$M, i, \pi \models p(x)$ iff $\pi(x) \in I_i(p)$

$M, i, \pi \models x = y$ iff $\pi(x) = \pi(y)$

$M, i, \pi \models \neg\alpha$ iff $M, i, \pi \not\models \alpha$

$M, i, \pi \models \alpha \vee \beta$ iff $M, i, \pi \models \alpha$ or $M, i, \pi \models \beta$

$M, i\pi \models \bigcirc\alpha$ iff $M, i+1, \pi \models \alpha$

$M, i, \pi \models \alpha\mathbf{U}\beta$ iff there exists $j \geq i$ such that $M, j, \pi \models \beta$ $\forall j'$, $i \leq j' < j$, $M, \pi, j' \models \alpha$.

$M, i, \pi \models (\exists x)\alpha$ iff there exists $a \in D$ such that $M, i, \pi[x \mapsto a] \models \alpha$

We can define derived boolean modalities $\wedge, \supset, \equiv$ in the usual way, as well as derived temporal modalities $\Box, \Diamond$ and the universally quantified formula $(\forall x)\alpha$.

We observe that with a single unary predicate *active* we can describe diverse properties of client server systems. When $active(x)$ holds at an instance i then it means a client with id x is active, that is, has been admitted in the client server system and getting service.

initially there are no active agents

$$(\forall x)\neg active(x)$$

at least two agents are active all the time

$$\Box\big((\exists x)(\exists y)\big[x \neq y \wedge active(x) \wedge active(y)\big]\big)$$

every active agent gets deactivated eventually

$$\Box\big((\forall x)\big[active(x) \supset active(x)\mathbf{U}\neg active(x)\big]\big)$$

there are two active agents which get deactivated simultaneously

$$\Box\big((\exists x)(\exists y)\big[active(x) \wedge active(y) \supset \Diamond(\neg active(x) \wedge \neg active(y))\big]\big)$$

every inactive agent gets activated eventually

$$\Box\big((\forall x)\big[\neg active(x) \supset \Diamond active(x)\big]\big)$$

at most one agent gets activated at each instance

$$\Box\big((\exists x)\big[\neg active(x) \supset \bigcirc active(x) \supset (\forall y)(y \neq x \supset (\neg active(y) \supset \bigcirc\neg active(y)))\big]\big)$$

5.2 Undecidability of $MFOTL$

Consider a fragment of $MFOTL$ with no equality (=) and $\mathbf{U}$ but containing $\Box$ and $\Diamond$. Let us call it $MFOTL^{-}$. We show $MFOTL^{-}$ to be undecidable, which in turn means that $MFOTL$ is undecidable too. This is done by encoding Minsky machines in $MFOTL^{-}$. The following discussion is from [70].

5.2.1 Minsky Machines

A two-counter or Minsky machine [69] is a well known Turing-complete formalism. A Minsky machine is an imperative program consisting of a sequence of labelled instructions $(l_1 : L_1); (l_2 : L_2); \ldots ; (l_m : L_m)$ which modify the values of two non-negative counters c_0 and c_1. The instructions, using counters c_n, for $n \in \{0, 1\}$ are of three kinds:

- $(l_i : HALT)$: Halts the machine.
- $(l_i : INC(c_n, l_j))$: Increments c_n and jumps to the instruction l_j.
- $(l_i : DEC(c_n, l_j, l_k))$: Tests if c_n is zero and jumps to the instruction l_j. If c_n is not zero then decrements c_n and jumps to l_k.

A configuration of a Minsky machine is a tuple (l_i, v_0, v_1), where l_i is the label of the next instruction to be executed and v_0 and v_1 are the current values of the counters. The moves between configurations are described by the reduction relation $\rightarrow_M$. $\rightarrow_M^*$ denotes the reflexive and transitive closure of $\rightarrow_M$.

Definition 5.2.1 (Reduction Relation)**.** *The reduction relation $\rightarrow_M$ over the set of configurations of a Minsky machine M is defined as follows:*

$M - INC$: *if $(l_i, INC(c_n, l_j))$ is an instruction in M and $v'_n = v_n + 1, v'_{1-n} = v_{1-n}$ then $(l_i, v_0, v_1) \rightarrow_M (l_j, v'_0, v'_1)$.*

$M - DEC$: *if $(l_i, DEC(c_n, l_j, l_k))$ is an instruction in M and $v_n \neq 0, v'_n = v_n - 1, v'_{1-n} = v_{1-n}$ then $(l_i, v_0, v_1) \rightarrow_M (l_k, v'_0, v'_1)$.*

$M - DECJ$: *if $(l_i, DEC(c_n, l_j, l_k))$ is an instruction in M and $v_n \neq 0$ then $(l_i, v_0, v_1) \rightarrow_M (l_j, v_0, v_1)$.*

We assume that counters are initially set to zero and the machine starts at the instruction l_1. That is, the initial configuration of any Minsky machine is of the form $(l_1, 0, 0)$. We say that a Minsky machine M halts if the control reaches the location of a HALT instruction.

Definition 5.2.2. *(Minsky Machine Computations) Let M be a Minsky machine with instructions $(l_1 : L_1); (l_2 : L_2); \ldots; (l_j : HALT); \ldots; (l_m : L_m)$. Let $\rightarrow_M$ be as defined above. We say that M* halts *if there exists a derivation $(l_j, v_0, v_1) \rightarrow^*_M (l_j, v_0, v_1) \not\rightarrow_M$.*

5.2.2 Encoding Minsky machines into $MFOTL^-$

In this section we show that given any Minsky machine M, we can effectively construct a formula φ_M that faithfully describes the behaviour of M. We shall assume a first-order signature with monadic predicates $\mathbf{out}(\cdot)$ and $\mathbf{not-zero}(\cdot)$. Furthermore, we assume the availability of propositional variables $\mathbf{isz}_n$, $\mathbf{inc}_n$, $\mathbf{dec}_n$, $\mathbf{idle}_n$, $\mathbf{zero}_n$, for $n \in \{0, 1\}$, and $\mathbf{halt}$.

The behaviour of any Minsky machine M can be simulated by the formula $\varphi_M = \Box(\varphi_{zero_0} \land \varphi_{zero_1} \land \varphi_{not-zero_0} \land \varphi_{not-zero_1} \land \varphi_{ins})$ where for $n \in \{0, 1\}$

$$\varphi_{zero_n}\colon\ \mathbf{zero}_n \supset \Big(\big(\mathbf{inc}_n \supset \varphi_{zero-inc_n}\big) \land \big(\mathbf{idle}_n \supset \varphi_{zero-idle_n}\big) \land \mathbf{isz}_n\Big)$$

$$\varphi_{zero-inc_n}\colon\ \bigcirc(\exists a)(\mathbf{not-zero}_n(a) \land \Box(\mathbf{out}(a) \supset \mathbf{zero}_n))$$

$$\varphi_{zero-idle_n}\colon\ \bigcirc\mathbf{zero}_n$$

$$\varphi_{not-zero_n}\colon\ (\forall x)\Big(\mathbf{not-zero}_n(x) \supset \big(\big(\mathbf{inc}_n \supset \varphi_{not-zero-inc_n}(x)\big) \land \big(\mathbf{dec}_n \supset \varphi_{not-zero-dec_n}(x)\big) \land \big(\mathbf{idle}_n \supset \varphi_{not-zero-idle_n}(x)\big) \land \neg\mathbf{isz}_n\big)\Big)$$

$$\varphi_{not-zero-inc_n}(x)\colon\ \bigcirc(\exists b)\big(\mathbf{not-zero}_n(b) \land \Box(\mathbf{out}(b) \supset \mathbf{not-zero}_n(x))\big)$$

$$\varphi_{not-zero-dec_n}(x)\colon\ \bigcirc\mathbf{out}(x)$$

$$\varphi_{not-zero-idle_n}(x)\colon\ \bigcirc\mathbf{not-zero}_n(x)$$

and

$$\varphi_{ins}\colon\ \bigwedge_{1 \le i \le m} \big(\mathbf{out}(l_i) \supset \varphi_{l_i:L_i}\big) \text{ where}$$

$$\varphi_{l_i:HALT}\colon\ \mathbf{halt}$$

$$\varphi_{l_i:INC(c_n,l_j)}\colon\ \neg\mathbf{halt} \land \mathbf{inc}_n \land \neg\mathbf{idle}_n \land \neg\mathbf{idle}_{1-n} \land \bigcirc\mathbf{out}(l_j)$$

$$\varphi_{l_i:DEC(c_n,l_j,l_k)}\colon\ \big(\mathbf{isz}_n \supset (\mathbf{idle}_n \land \bigcirc\mathbf{out}(l_j))\big) \land \big(\neg\mathbf{isz}_n \supset (\neg\mathbf{idle}_n \land \mathbf{dec}_n \land \bigcirc\mathbf{out}(l_k))\big) \land \big(\mathbf{idle}_{1-n} \land \neg\mathbf{halt}\big)$$

We briefly describe the intuition behind formulas here. The formulae modelling the counters c_0 and c_1 are obtained by replacing the subscript n by 0 and 1, respectively, in the formulae φ_{zero_n} and $\varphi_{not-zero_n}$. The formula φ_{zero_n} models the state $c_n = 0$ and $\varphi_{not-zero_n}$ models the state $c_n = k$ for $k > 0$.

Once $\mathbf{zero}_n$ holds, $\mathbf{isz}_n$ must also hold. We can then use the proposition $\mathbf{isz}_n$ to test if the counter c_n is zero.

If the current instruction does not modify the value of c_n then $\mathbf{idle}_n$ must hold and then, $\bigcirc\mathbf{zero}_n$ must hold.

When an increment instruction is executed, $\mathbf{inc}_n$ holds and so does a formula of the form $\beta = \bigcirc(\exists a)\big(\mathbf{not} - \mathbf{zero}_n(a) \wedge \Box(\mathbf{out}_n)(a) \supset \alpha\big)$. In β, α is $\mathbf{zero}_n$ if $c_n = 0$ and $\mathbf{not} - \mathbf{zero}_n(x)$ otherwise. Intuitively, α represents the state immediately before the last increment instruction took place. This way, when a decrement operation is performed $\mathbf{out}(a)$ holds and so does α.

Consider $\varphi_{not-zero_n}$ which is of the form $(\forall x)\big(\mathbf{not} - \mathbf{zero}_n(x) \supset \gamma\big)$. As mentioned previously, a formula of the form β holds when an increment action is performed. Using β in conjunction with $\varphi_{not-zero_n}$ we obtain an instantiation of the form $(\exists a)\gamma[a/x]$ that represents the state $c_n = k+1$. Notice that when $(\exists a)\gamma[a/x]$ holds, $\mathbf{isz}_n$ must not hold. Furthermore, if the counter is not modified by the current instruction ($\mathbf{idle}_n$ holds), $\mathbf{not} - \mathbf{zero}_n(a)$ must hold and then, the counter takes the same value in the next instant.

For the set of instructions $(l_1 : L_1)(l_2 : L_2)\cdots(l_m : L_m)$ we assume a set of variables $l_1, l_2, \cdots, l_m$. If the predicate $\mathbf{out}(l_i)$ holds in a state, it means that the instruction l_i is executed. In the case of halt instruction $(l_i, HALT)$, $\mathbf{halt}$ holds, whereas for increment and decrement instructions $\neg\mathbf{halt}$ holds.

The formula representing an increment operation $(l_i : INC(c_n, l_j))$ assures that $\mathbf{inc}_n$ holds and $\mathbf{idle}_{1-n}$ holds while $\mathbf{idle}_n$ does not hold.

The formula representing a decrement instruction $(l_i : DEC(c_n, l_j, l_k))$ tests if the counter c_n is zero via the proposition $\mathbf{isz}_n$. If this is the case, then it activates the instruction l_j in the next time instant via $\mathbf{out}$. Otherwise, $\mathbf{dec}_n$ must hold and $\mathbf{out}(l_k)$ must hold in the next time instant.

Encoding of Numbers and Configurations

In order to show that φ_M faithfully describes the behaviour of the Minsky machine M, we should first give a suitable representation of numbers (which are possible values of the counters) and configurations of M.

As already mentioned, when an increment operation is performed, a formula of the form

$$\beta = \bigcirc(\exists a)\big(\mathbf{not}-\mathbf{zero}_n(a) \wedge \Box(\mathbf{out}_n)(a) \supset \alpha\big)$$

must hold, where α represents the state immediately before the last increment instruction took place. A decrement operation causes that $\mathbf{out}_n(a)$ holds and so does α. We can then represent the state $c_n = k$, for $k > 0$ and $n \in \{0,1\}$, as a formula $\varphi_{c_n=k}$ of the form $(\exists a_1)(\exists a_2)\cdots(\exists a_k)\big(\alpha_1 \wedge \alpha_2 \wedge \cdots \wedge \alpha_k \wedge \mathbf{not}-\mathbf{zero}_n(a_k)\big)$ where

- $\alpha_1 = \Box(\mathbf{out}_n(a_1) \supset \mathbf{zero}_n)$ and
- $\alpha_i = \Box(\mathbf{out}_n(a_i) \supset \mathbf{not}-\mathbf{zero}_n(a_{i-1}))$, for $1 < i \leq k$.

Clearly, $\varphi_{c_n=0} = \mathbf{zero}_n$, for $n \in \{0,1\}$. Now, using the previous definition of numbers, we can define the $MFOTL^-$ formula representing a configuration of a Minsky machine.

Definition 5.2.3. *Let M be a Minsky machine with instructions $(l_1 : L_1)(l_2 : L_2)\cdots(l_m : L_m)$. Then, a configuration (l_i, v_0, v_1) of M can be represented as follows:*

$$\varphi_{(l_i,v_0,v_1)} = \varphi_M \wedge \varphi_{c_0=v_0} \wedge \varphi_{c_1=v_1} \wedge \mathbf{out}(l_i).$$

Now, we can use the above encoding to exhibit a formula that is valid if and only if the machine M loops. This allows us to conclude that the validity problem in $MFOTL^-$ is undecidable.

We can verify that φ_M faithfully describes the computations of M.

Lemma 5.2.4 ([70])**.** *Let M be a Minsky machine with instructions $(l_1 : L_1)(l_2 : L_2)\cdots(l_m : L_m)$. Then, for any pair of configurations (l_i, v_0, v_1) and (l'_i, v'_0, v'_1) of M following holds true:*

If $(l_i, v_0, v_1) \to_M (l'_i, v'_0, v'_1)$ then $\varphi_{(l_i,v_0,v_1)} \models \neg\mathbf{halt} \wedge \bigcirc\varphi_{(l'_i,v'_0,v'_1)}$

Furthermore, if l_i is a HALT instruction, i.e., $(l_i, v_0, v_1) \not\to_M$ then the following holds: $\varphi_{(l_i,v_0,v_1)} \models \textbf{halt}$.

Using the previous lemma, it can now be shown that a machine M produces an infinite run if and only if the formula $\varphi_M \supset \Box\neg\textbf{halt}$ is valid.

Lemma 5.2.5 ([70])**.** *A Minsky machine M loops if and only if*

$$\varphi_{(l_i,v_0,v_1)} \models \Box\neg\textbf{halt}.$$

The above lemma leads us directly to the following theorem:

Theorem 5.2.6. *The validity problem (hence, satisfiability problem) in $MFOTL^-$ is undecidable.*

To conclude, we saw that $MFOTL$ is too expressive to be used as specification language, say for client server systems with unbounded agents. As pointed out in the beginning, we need to limit the expressiveness of the logic in some ways to obtain decidable satisfiability problem and therefore a decidable verification problem. As the reader may have observed, undecidability in $MFOTL$, essentially, happens due to unrestrained application of quantification over temporal modalities.

In the next chapter, we present two many-sorted $MFOTL$ fragments, one with no free variable in the scope of temporal subformulas and another with at most one free variable and show their decidability. We also show, with examples, how they can be used to specify the client server systems SPS and SAS, already described in Chapter 4.

6

Decidable Logics for Temporal Specifications of Client-Server Systems

The undecidability of $MFOTL$ arises due to quantification over temporal modalities. The obvious way to obtain decidability would be to put constraints on the quantifier-modality combination. On the most superficial level, we can get two decidable fragments of $MFOTL$ as follows:

1. $MFOTL$ fragment with no free variables in the scope of temporal modalities ($\Diamond/\Box$ and $\bigcirc$). In this case, we take MFO sentences and close them with respect to temporal modalities.

2. $MFOTL$ fragment with at most one free variable in the scope of temporal modalities. This is referred to as monadic monodic temporal logic in the literature [45].

Another decidable fragment of $MFOTL$, with constraints orthogonal to those stated above, is the one-variable fragment [38],[80]. In this case, there are no restrictions over the quantifiers, modalities and their combinations but, we can use no more than one variable in any formula. Clearly, if we allow two variables freely, we shall again be able to encode the $\mathbb{N} \times \mathbb{N}$ recurring tiling problem.

In this chapter, we propose two decidable $MFOTL$ fragments, one of the first type and another of the second type, meant to specify SPS and SAS respectively. The decidability argument for the satisfiability of formulae in each logic crucially uses the formula automaton construction, as first proposed in [84].

6.1 $\mathcal{L}_{SPS}$

In this section we describe a logical language to specify and verify SPS-like systems. Such a language has two mutually exclusive dimensions. One, captured by MFO fragment, talking about the plurality of clients asking for a variety of services. The other, captured by LTL fragment, talks about the temporal variations of services being rendered. Furthermore, the MFO fragment has to be multi-sorted to cover the multiplicity of service types. Keeping these issues in mind, we frame a logical language, which we call $\mathcal{L}_{SPS}$, a combination of LTL and multi-sorted MFO. In the case of LTL, atomic formulae are propositional constants which have no further structure. In $\mathcal{L}_{SPS}$, there are two kind of atomic formulae, basic server properties from P_s, and MFO-sentences over client properties P_c. Consequently, these formulae are interpreted over sequences of MFO-structures juxtaposed with LTL-models.

6.1.1 The Logic $\mathcal{L}_{SPS}$

At the outset, we fix Γ_0, a finite set of client types. The set of client formulae are defined over a countable set of atomic client predicates P_c, which are composed of disjoint predicates P_c^u of type u , for each $u \in \Gamma_0$. Also, let Var be a countable supply of variable symbols and CN be a countable set of client names. CN is divided into disjoint sets of types from Γ_0 via $\lambda : CN \to \Gamma_0$. Similarly, Var is divided using $\Pi : Var \to \Gamma_0$. We use x, y to denote elements in Var and a, b for elements in CN.

Formally, the set of client formulae Φ is:

$$\alpha, \beta \in \Phi ::= p(x : u), p \in P_c^u \mid x = y, x, y \in Var_u \mid \neg\alpha \mid \alpha \vee \beta \mid (\exists x : u)\alpha$$

Let $\mathcal{S}_\Phi$ be the set of all sentences in Φ, then, the Server formulae are defined as follows:

$$\psi \in \Psi ::= q \in P_s \mid \varphi \in \mathcal{S}_\Phi \mid \neg\psi \mid \psi_1 \vee \psi_2 \mid \bigcirc\psi \mid \psi \mathbf{U} \psi'$$

6.1.2 Semantics

This logic is interpreted over sequences of MFO models composed with LTL models. Formally, a model is a triple $M = (\nu, D, I)$ where

1. $\nu = \nu_0\nu_1\cdots$, where $\forall i \in \omega$, $\nu_i \subset_{fin} P_s$, gives the local properties of the server at instance i,
2. $D = D_0D_1D_2\cdots$, where $\forall i \in \omega$, $D_i = (D_i^u)_{u\in\Gamma_0}$ where $D_i^u \subset_{fin} CN_u$, gives the identity of the clients of each type being served at instance i and
3. $I = I_0I_1I_2\cdots$, where $\forall i \in \omega$, $I_i = (I_i^u)_{u\in\Gamma_0}$ and $I_i^u : D_i^u \to 2^{P_c^u}$ gives the properties satisfied by each live agent at ith instance, in other words, the corresponding states of live agents.
   Alternatively, I_i^u can be given as $I_i^u : D_i^u \times P_c^u \to \{\top, \bot\}$, an equivalent form.

Satisfiability Relations $\models$, $\models_\Phi$

Let $M = (\nu, D, I)$ be a valid model and $\pi : Var \to CN$ be a partial map consistent with respect to λ and Π. Then, the relations $\models$ and $\models_\Phi$ can be defined, via induction over the structure of ψ and α, respectively, as follows:

1. $M, i \models q$ iff $q \in \nu_i$.
2. $M, i \models \varphi$ iff $M, \emptyset, i \models_\Phi \varphi$.
3. $M, i \models \neg\psi$ iff $M, i \not\models \psi$.
4. $M, i \models \psi \vee \psi'$ iff $M, i \models \psi$ or $M, i \models \psi'$.
5. $M, i \models \bigcirc\psi$ iff $M, i+1 \models \psi$.
6. $M, i \models \psi\mathbf{U}\psi'$ iff $\exists j \geq i$, $M, j \models \psi'$ and $\forall i' : i \leq i' < j$, $M, i' \models \psi$.

7. $M, \pi, i \models_\Phi p(x : u)$ iff $\pi(x) \in D_i^u$ and $I_i(\pi(x), p) = \top$.
8. $M, \pi, i \models_\Phi x = y$ iff $\pi(x) = \pi(y)$.
9. $M, \pi, i \models_\Phi \neg\alpha$ iff $M, \pi, i \not\models_\Phi \alpha$.

10. $M, \pi, i \models_\Phi \alpha \vee \beta$ iff $M, \pi, i \models_\Phi \alpha$ or $M, \pi, i \models_\Phi \beta$.

11. $M, \pi, i \models_\Phi (\exists x : u)\alpha$ iff $\exists a \in D_i^u$ and $M, \pi[x \mapsto a], i \models_\Phi \alpha$.

6.2 Specification Examples Using $\mathcal{L}_{SPS}$

In this section, we would like to show that our logic $\mathcal{L}_{SPS}$ adequately captures many of the facets of SPS-like systems. We consider the Loan Approval Web Service, which has already been explained with a number of examples, and frame specifications to demonstrate the use of $\mathcal{L}_{SPS}$.

In a Loan Approval System, clients (customers) apply for loans of different sizes and wait for the appropriate response from the server (loan officer). The client with a request for a particular loan amount can be seen as a client of *that* type. Therefore, we can have client types as say, $\Gamma_0 = \{h, l, m\}$ and client properties as $P_c = \{req_h, req_l, ans_h, ans_l, req_m, ans_m\}$. Here h means a loan request of type (size) high, l means a loan request of type (size) low and m means a loan request of type (size) medium. Now, we can write a few simple specifications in $\mathcal{L}_{SPS}$ as follows:

1. $\psi_0 = \neg\big((\exists x : h)req_h(x) \vee (\exists x : l)req_l(x) \vee (\exists x : m)req_m(x)\big)$
   which means initially there are no pending requests.

2. $\psi_1 = \Box[(\exists x : l)req_l(x) \supset \bigcirc(\exists y : l)ans_l(y)]$
   which means whenever there is a request of type *low* there is an approval for type *low* in the next instant.

3. $\psi_2 = \Box[(\exists x : h)req_h(x) \supset \neg(\exists y : l)req_l(y)]$
   which means there is no request of type *low* taken up as long as there is a *high* request pending.

4. $\psi_3 = \Box[(\exists x : l)req_l(x) \vee (\exists y : h)req_h(y) \vee (\exists z : m)req_m(z)]$
   which means there is at least one request of each type pending all the time.

5. $\psi_4 = \Box[(\exists x : h)req_h(x) \supset \neg[(\exists y : l)req_l(y) \vee (\exists y : l)req_l(y)]]$ which is similar to ψ_2, there are no pending medium or low requests with a high request.

Note that none of these formulae make use of equality (=) predicate. Using =, we can make stronger statements as follows:

1. $\psi_5 = \Box[(\exists x : h)req_h(x) \wedge (\forall y : h)\big(req_h(y) \supset x = y\big)]$
   which means at all times there is *exactly one* pending request of type *high*.

2. $\psi_6 = \Box[\big(\neg(\exists x : h)req_h(x)\big) \vee \Big((\exists x : h)req_h(x) \wedge (\forall y : h)\big(req_h(y) \supset x = y\big)\Big)]$
   which means at all times there is *at most* one pending request of type *high*.

In the same vein, using =, we can count the requests of each type and say more interesting things. For example, if $\varphi_h^2 = (\exists x : h)(\exists y : h)(\exists z : h)\big(req_h(x) \wedge req_h(y) \wedge req_h(z) \supset (x = y \vee y = z)\big)$ asserted at a point means there are at most 2 requests of type h pending then we can frame the following formula:

- $\psi_5 = \Box(\varphi_h^2 \supset \bigcirc(\varphi_h^2 \supset \Box\varphi_h^2))$
  which means, if there are at most two pending requests of type *high* at successive instants then thereafter the number stabilizes.

Unfortunately, owing to a lack of provision for free variables in the scope of temporal modalities, we can't write specifications which seek to match requests and approvals. Here is a sample.

$$\Box((\forall x)req_u(x) \supset \bigcirc\Diamond ans_u(x))$$

which means, if there is a request of type u at some point of time then the **same** is approved some time in future.

The challenge is to come up with appropriate constraints on specifications which allow us to express interesting properties as well as remain decidable to verify.

6.3 Satisfiability of $\mathcal{L}_{SPS}$

We settle the satisfiability issue for $\mathcal{L}_{SPS}$ using the automata theoretic techniques, first proposed by Vardi and Wolper [84]. That is, given ψ_0, an $\mathcal{L}_{SPS}$-formula, we compute a formula automaton A_{ψ_0}, such that the following holds.

Theorem 6.3.1. *ψ_0 is satisfiable iff $Lang(A_{\psi_0})$ is non-empty.*

Notice that the statement does not guarantee an equivalence between models of ψ_0 and language of A_{ψ_0}. This happens because the models of ψ_0 would invariably be described over possibly infinite domains whereas the automaton shall have finite number of states in order to have decidable reachability properties. Assuming the theorem, we see that satisfiability of ψ_0 can be checked in time linear in the size of A_{ψ_0}. We shall see later that the size of the formula automaton crucially depends on the number of monadic predicates occurring in ψ_0, k, and the number of variable symbols, r.

6.3.1 Closure Sets

Before we set off defining appropriate closure sets for ψ_0, we rename variables in ψ_0 so that none of them are reused. Thereafter we define a number of subformula closed sets, CL^Ψ and T_φ, for every MFO sentence $\varphi \in CL^\Psi$. CL^Ψ is the smallest set containing ψ_0 and satisfying the following conditions:

1. $\psi_0 \in CL^\Psi$.
2. if $\neg\psi \in CL^\Psi$ then $\psi \in CL^\Psi$.
3. if $\psi_1 \vee \psi_2 \in CL^\Psi$ then $\psi_1, \psi_2 \in CL^\Psi$.
4. if $\bigcirc\psi \in CL^\Psi$ then $\psi \in CL^\Psi$.
5. if $\psi_1\mathbf{U}\psi_2 \in CL^\Psi$ then $\psi_1, \psi_2, \bigcirc(\psi_1\mathbf{U}\psi_2) \in CL^\Psi$.

cl^Ψ is obtained by closing CL^Ψ with respect to $\neg$.

$$cl^\Psi \overset{\text{def}}{=} CL^\Psi \cup \{\neg\psi \mid \psi \in CL^\Psi \text{ taking } \neg\neg\psi \overset{\text{def}}{=} \psi\}$$

For each $\varphi \in CL^\Psi$, we define T_φ as the smallest set containing φ and satisfying the following conditions:

1. if $\varphi \in CL^\Psi$ then $\varphi \in T_\varphi$.
2. if $\neg\alpha \in T_\varphi$ then $\alpha \in T_\varphi$.
3. if $\alpha \vee \beta \in T_\varphi$ then $\alpha, \beta \in T_\varphi$.

4. if $(\exists x : u)\alpha \in T_\varphi$ then $\alpha \in T_\varphi$ and $x \in Var_u(\psi_0)$.

For every $u \in \Gamma_0$, cl^X_u is a closure set from which we define partitions of $Var_u(\psi_0)$, which, in turn, give all possible valuation skeletons for variables of type u.

- for every $x, y \in Var_u(\psi_0)$, $x = y, x \neq y, y = x, y \neq x \in cl^X_u$.

Over the whole $Var(\psi_0)$, we have $\bigcup\limits_{u\in\Gamma_0} cl^X = cl^X_u$. Let $T = \bigcup\limits_{\varphi\in cl^\Psi} T_\varphi$. Now, the set of monadic predicates of type $u \in \Gamma_0$ occurring in ψ_0 are $Prop_u(\psi_0) = \{p \in Prop_u \mid p(x : u) \in T$ for some $x \in Var_u\}$. Let $Prop(\psi_0) = \bigcup\limits_{u\in\Gamma_0} Var_u(\psi_0)$. Let the set of variables occurring in ψ_0 be $Var(\psi_0) = \bigcup\limits_{u\in\Gamma_0} Var_u$. Also, let $Q_{\psi_0} = \{q_1, \cdots, q_{k_Q}\}$ be the local propositions occurring in ψ_0. Let, $|Var(\psi_0)| = r$ and $|Prop(\psi_0)| = k_P$.

$\mathbb{D}$ is the set of all legitimate valuations constructed from $Prop(\psi_0)$. In the case of multi-sorted MFO, for each type $u \in \Gamma_0$, we define $\mathbb{D}_u = 2^{Prop_u(\psi_0)}$. Then, $\mathbb{D} = \bigcup\limits_{u\in\Gamma_0} \mathbb{D}_u$. For every $\mathfrak{p} \in \mathbb{D}$ and $1 \leq s \leq r$ we can define $\mathcal{D}^s_{\mathfrak{p}} = \{(\mathfrak{p}, s') \mid 1 \leq s' \leq s\}$. Now, for every $\mathfrak{p} \in \mathbb{D}$, define $\mathcal{D}_{\mathfrak{p}} = \{\mathcal{D}^s_{\mathfrak{p}} \mid 1 \leq s \leq r\} \cup \{\emptyset\}$. Next, define $\mathcal{D} = \Pi_{\mathfrak{p}}\mathcal{D}_{\mathfrak{p}}$. Now, for every $d \in \mathcal{D}$, we define $\mathfrak{d} = \bigcup\limits_{\mathfrak{p}} d_{\mathfrak{p}}$. Now, we define $\mathfrak{D} = \{\mathfrak{d} \mid d \in \mathcal{D}\}$. Clearly, from any $\mathfrak{d}$, we can extract the part corresponding to a type that is, $\mathfrak{d}_u$. We omit the details.

For a given $\mathfrak{d} \in \mathfrak{D}$ we can define an induced MFO-model as follows: $\mathfrak{m} = (\mathfrak{d}, \iota)$ where $\iota : \mathfrak{d} \times Prop(\psi_0) \to \{\top, \bot\}$ such that for every $\mathfrak{a} \in \mathfrak{d}$ and $p \in Prop(\psi_0)$,

$$\iota(\mathfrak{a}, p) = \begin{cases} \top, & \text{if } p \in \mathfrak{a}[1]. \\ \bot, & \text{if } p \notin \mathfrak{a}[1]. \end{cases}$$

Now, for every $\mathfrak{d} \in \mathfrak{D}$, we define a closure set $CL(\Phi, \mathfrak{d})$ as the smallest set satisfying following conditions.

1. for every $\varphi \in cl^\Psi$, $\varphi \in CL(\Phi, \mathfrak{d})$.

2. if $\neg\alpha \in CL(\Phi, \mathfrak{d})$ then $\alpha \in CL(\Phi, \mathfrak{d})$.

3. if $\alpha \vee \beta \in CL(\Phi, \mathfrak{d})$ then $\alpha, \beta \in CL(\Phi, \mathfrak{d})$.

4. if $(\exists x)\alpha \in CL(\Phi, \mathfrak{d})$ then for every $\mathfrak{a} = (\mathfrak{p}, l) \in \mathfrak{d}$, $\alpha[\mathfrak{a}/x] \in CL(\Phi, \mathfrak{d})$.

Now, we close the set with respect to negation, as follows:

$$cl(\Phi,\mathfrak{d}) = CL(\Phi,\mathfrak{d}) \cup \{\neg\alpha \mid \alpha \in CL(\Phi,\mathfrak{d}) \text{ taking } \neg\neg\alpha \overset{\text{def}}{=} \alpha\}.$$

6.3.2 Atoms

We define three kind of atoms and combine them consistently to construct global atoms which become part of states.

For each type $u \in \Gamma_0$, $At_u^X \subseteq cl_u^X$ is an (X,u)-atom when following conditions hold:

1. for every $x \in Var_u(\psi_0)$, $x = x \in At_u^X$.

2. for every $x, y \in Var_u(\psi_0)$, $x = y \in At_u^X$ iff $y = x \in At_u^X$.
   Conversely, for every $x, y \in Var_u(\psi_0)$, $x \neq y \in At_u^X$ iff $y \neq x \in At_u^X$.

3. for every $x, y, z \in Var_u(\psi_0)$, if $x = y, y = z \in At_u^X$ then $x = z \in At_u^X$.

The full X-atom is $At^X = \bigcup\limits_{u\in\Gamma_0} At_u^X$. With the given set of equality conditions, let $\equiv_{At^X}$ be the corresponding equivalence relation and C_{At^X} be the induced partition of $Var(\psi_0)$.

$At^\Psi \subseteq cl^\Psi$ is an Ψ-atom when following conditions hold.

1. for every $\psi \in cl^\Psi$, $\psi \in At^\Psi$ iff $\neg\psi \notin At^\Psi$.

2. for every $\psi_1 \vee \psi_2 \in cl^\Psi$, $\psi_1 \vee \psi_2 \in At^\Psi$ iff $\psi_1 \in At^\Psi$ or $\psi_2 \in At^\Psi$.

3. for every $\psi_1 \mathbf{U} \psi_2 \in At^\Psi$, $\psi_1 \mathbf{U} \psi_2 \in At^\Psi$ iff $\psi_2 \in At^\Psi$ or $\psi_1, \bigcirc(\psi_1 \mathbf{U} \psi_2) \in At^\Psi$.

For any $\mathfrak{d}$, $At^{\Phi,\mathfrak{d}} \subseteq cl(\Phi,\mathfrak{d})$ is an $(\Phi,\mathfrak{d})$-atom when following conditions hold.

1. for every $\alpha \in cl(\Phi,\mathfrak{d})$, $\neg\alpha \in AT^{\Phi,\mathfrak{d}}$ iff $\alpha \notin At^{\Phi,\mathfrak{d}}$.

2. for every $\alpha \vee \beta \in cl(\Phi,\mathfrak{d})$, $\alpha \vee \beta \in At^{\Phi,\mathfrak{d}}$ iff $\alpha \in At^{\Phi,\mathfrak{d}}$ or $\beta \in At^{\Phi,\mathfrak{d}}$.

3. for every $(\exists x : u)\alpha \in cl(\Phi,\mathfrak{d})$, $(\exists x : u)\alpha \in At^{\Phi,\mathfrak{d}}$ iff $\bigvee_{\mathfrak{a}\in\mathfrak{d}_u} \alpha[\mathfrak{a}/x]$.

4. for every $p \in Prop(\psi_0)$, for every $\mathfrak{a} = (\mathfrak{p}, l) \in \mathfrak{d}$, $p \in \mathfrak{a}[1]$ iff $p(\mathfrak{a}) \in At^{\Phi,\mathfrak{d}}$.
   Conversely, for every $p \in Prop(\psi_0)$, for every $\mathfrak{a} \in \mathfrak{d}$, $p \notin \mathfrak{a}[1]$ iff $\neg p(\mathfrak{a}) \in At^{\Phi,\mathfrak{d}}$.

5. for every $\mathfrak{a}, \mathfrak{b} \in \mathfrak{d}$, for every $(\exists x : u)\alpha \in cl(\Phi, \mathfrak{d})$ if $\mathfrak{a} = \mathfrak{b} \in At^{\Phi,\mathfrak{d}}$ then $\alpha[\mathfrak{a}/x] \in At^{\Phi,\mathfrak{d}}$ iff $\alpha[\mathfrak{b}/x] \in At^{\Phi,\mathfrak{d}}$

6. for every $\mathfrak{a}, \mathfrak{b} \in \mathfrak{d}$, $(\mathfrak{a} = \mathfrak{b}) \in At^{\Phi,\mathfrak{d}}$ iff $\mathfrak{a} = \mathfrak{b}$.

   Note: if $\mathfrak{a} = (\mathfrak{p}, l)$ and $\mathfrak{b} = (\mathfrak{p}', l')$ then $\mathfrak{a} = \mathfrak{b}$ iff $\mathfrak{p} = \mathfrak{p}'$ and $l = l'$ also, $\mathfrak{a} \neq \mathfrak{b}$ iff $\mathfrak{p} \neq \mathfrak{p}'$ or $l \neq l'$

Let $\mathbf{AT}^X$ be the set of all legal X-atoms, $\mathbf{AT}^\Psi$ be the set of all legal Ψ-atoms and $\mathbf{AT}^{\Phi,\mathfrak{d}}$ be the set of all $(\Phi, \mathfrak{d})$-atoms as defined above. Furthermore, let $\mathbf{AT}^\Phi = \bigcup_{\mathfrak{d} \in \mathfrak{D}} \mathbf{AT}^{\Phi,\mathfrak{d}}$ be the set of all legal Φ-atoms. Now, we define atoms proper.
For any $At^X \in \mathbf{AT}^X$, $At^\Psi \in \mathbf{AT}^\Psi$, and $At^{\Phi,\mathfrak{d}} \in At^\Phi$, for some $\mathfrak{d} \in \mathfrak{D}$, $At = At^X \cup At^\Psi \cup At^{\Phi,\mathfrak{d}}$ is a global atom if following conditions hold.

1. C_{At^X} is consistent with $\mathfrak{d}$. That is, $|C_{At^X}| \leq| \mathfrak{d}|$.

2. for every φ, $\varphi \in At^\Psi$ iff $\varphi \in At^{\Phi,\mathfrak{d}}$.

For a given global atom $At = At^X \cup At^\Psi \cup At^{\Phi,\mathfrak{d}}$ we define two associated sets $(\exists X)(At)$ and $(\forall X)(At)$ as follows: $(\exists X)(At) = \{x \in Var(\psi_u) \mid (\exists x : u)\alpha \in At^{\Phi,\mathfrak{d}}$ for some u and $\alpha\}$ and $(\forall X)(At) = Var(\psi) - (\exists X)(At)$. $(\exists X)(At)$ is the set of all those variables which occur existentially in At and $(\forall X)(At)$ contains the rest. Clearly, due to the prior renaming of variables in ψ_0, any variable $x \in Var(\psi_0)$ can occur either in its existential form or universal form but never both.

Let $\mathbf{AT}$ be the set of all legal global atoms. Also, let $UR = \{\psi_1 \mathbf{U} \psi_2 \in cl^\Psi\}$ be the set of all $\mathbf{U}$-requirements in cl^Ψ. Now, we give the definition of formula automaton corresponding to ψ_0.

Definition 6.3.2. *The* formula automaton *A_{ψ_0}, corresponding to the given formula ψ_0, is a four-tuple $(Q, \longrightarrow, I, G)$ where*

1. $Q = \mathbf{AT} \times 2^{UR}$.

2. $I \subseteq Q = \{(At, un) \mid \psi_0 \in At, un = \emptyset\}$.

3. $G \subseteq Q = \{(At, un) \mid un = \emptyset\}$.

4. $\longrightarrow \subseteq Q \times \Sigma \times Q$ *such that* $(At, un) \xrightarrow{(\mathfrak{q},\mathfrak{m})} (At', un')$ *when following conditions hold:*

(a) $\mathfrak{q} = At \cap Q_{\psi_0}$.

(b) $\mathfrak{m}$ *is the MFO-model induced by* $\mathfrak{d}$*, where* $At = At^X \cup At^\Psi \cup At^{\Phi,\mathfrak{d}}$.

(c) for every $\bigcirc\psi \in cl^\Psi$, $\bigcirc\psi \in A$ *iff* $\psi \in At'$.

(d) The set un' *is defined as follows:*

$$un' = \begin{cases} \{\psi_1 \mathbf{U} \psi_2 \in At' \mid \psi_2 \notin At'\}, & \textit{if } un = \emptyset. \\ \{\psi_1 \mathbf{U} \psi_2 \in un \mid \psi_2 \notin At'\}, & \textit{otherwise.} \end{cases}$$

6.3.3 Correctness

We first prove the ($\Leftarrow$) direction of theorem 6.3.1.

Lemma 6.3.3. *If* $Lang(A_{\psi_0})$ *is non-empty then* ψ_0 *is satisfiable in* $\mathcal{L}_{SPS}$.

Proof. Let $\mathfrak{w} \in Lang(A_{\psi_0})$ such that ρ is a successful run of A_{ψ_0} over $\mathfrak{w}$. Let $\rho = (At_0, un_0)(At_1, un_1)(At_2, un_2)\cdots$ and $\mathfrak{w} = (\mathfrak{q}_0, \mathfrak{m}_0)(\mathfrak{q}_1, \mathfrak{m}_1)(\mathfrak{q}_2, \mathfrak{m}_2)\cdots$. We know that $\forall i \in \omega$, $At_i = At_i^X \cup At_i^\Psi \cup At^{\Phi,\mathfrak{d}_i}$, where $\mathfrak{d}_i \in \mathfrak{D}$. For all $i \in \omega$, $\mathfrak{m}_i$ is the MFO structure induced by $\mathfrak{d}_i$. $\mathfrak{m}_i = (\mathfrak{d}_i, \iota_i)$ such that $\iota_i : \mathfrak{d}_i \times Prop(\psi_0) \to \{\top, \bot\}$ where $\forall \mathfrak{a} \in \mathfrak{d}_i, \forall p \in Prop(\psi_0)$, $\iota_i(\mathfrak{a}, p) = \top$ if $p \in \mathfrak{a}[1]$ else $\iota_i(\mathfrak{a}, p) = \bot$ if $p \notin \mathfrak{a}[1]$. The task to extract a model $M_\rho = (\nu_\rho, D_\rho, I_\rho)$ from the good run ρ such that $M_\rho, 0 \models \psi_0$ is similarly straightforward.

1. ν_ρ is the sequence $\mathfrak{q}_0\mathfrak{q}_1\mathfrak{q}_2\cdots$,
2. D_ρ is the sequence $\mathfrak{d}_0\mathfrak{d}_1\mathfrak{d}_2\cdots$ and
3. I_ρ is the sequence $\iota_0\iota_1\iota_2\cdots$.

Here is a preliminary claim which will be needed in the argument of the actual claim, the truth lemma.

Claim 6.3.4. $\forall i \in \omega$*,* $\forall \alpha \in cl^{\Phi,\mathfrak{d}_i}$*,* $\alpha \in At_i^{\Phi,\mathfrak{d}_i}$ *iff* $M_\rho, i \models_\Phi \alpha$.

The proof is by induction on the structure of α. The base case, the only interesting case, follows from the construction of the model above.
Here is the *Truth lemma.*

Claim 6.3.5. $\forall i \in \omega$*,* $\forall \psi \in cl^\Psi$*,* $\psi \in At_i^\Psi$ *iff* $M_\rho, i \models \psi$

The proof is by induction on the structure of ψ.

Assuming the Claim 6.3.5, since $\psi_0 \in At_0^{\Psi}$, $M_\rho, 0 \models \psi_0$. Hence, ψ_0 is satisfiable and Lemma 6.3.3 holds. □

Now, we come to the ($\Rightarrow$) direction of Theorem 6.3.1.

Lemma 6.3.6. *if ψ_0 is satisfiable in $\mathcal{L}_{SPS}$ then $Lang(A_{\psi_0})$ is non-empty.*

Proof. Let $M = (\nu, D, I)$ be a model for ψ_0, That is $M, 0 \models \psi_0$. We need to define a sequence $(At_0, un_0)(At_1, un_1)(At_2, un_2)\cdots$ such that $\forall i \in \omega$, $At_i = At_i^X \cup At_i^{\Psi} \cup At_i^{\Phi,\mathfrak{d}_i}$, for some $\mathfrak{d}_i \in \mathfrak{D}$ and

1. $At_i^X \subseteq cl^X$, $At_i^{\Psi} \subseteq cl^{\Psi}$ and $At_i^{\Phi,\mathfrak{d}_i} \subseteq cl(\Phi, \mathfrak{d}_i)$
2. At_i^{Ψ} and $At_i^{\Phi,\mathfrak{d}}$ satisfy all atom properties and
3. $\equiv_{At_i^X}$ is consistent with $\mathfrak{d}_i$.

At_i^{Ψ} can be computed without taking recourse to any finite model.

$$\forall \psi \in cl^{\Psi}, \psi \in At_i^{\Psi} \text{ iff } M, i \models \psi$$

Computing At_i^X and $At^{\Phi,\mathfrak{d}_i}$ is a bit tricky. We need to produce only one ρ, and that also over a single path in the legal π tree, at each $i \in \omega$, which can be traversed while satisfying the formula ψ_0.

1. Given $M = (D, I)$, compute the finite domain $\mathfrak{d} = \mathfrak{d}_0\mathfrak{d}_1\cdots$ and corresponding interpretation $\iota = \iota_0\iota_1\cdots$ as follows:

   For all $u \in \Gamma_0$, every $a \in D_u$ and $i \in \omega$, define $\sigma(a, i) = \{p \in P_u(\psi_0) \mid I_i(a, p) = \top\}$. Let $d_i = \{\sigma(a, i) \mid a \in D\}$, for every $i \in \omega$. Obviously, $d_i \in \mathbb{D}$ and $|d_i| \leq 2^k$. For every $i \in \omega$ and for every $\mathfrak{p} \in d_i$, compute $D_{\mathfrak{p}}^i = \{a \in D \mid \sigma(a, i) = \mathfrak{p}\}$. Now, for every $\mathfrak{p} \in d_i$ define $\mathfrak{n}_{\mathfrak{p}}^i$ as follows:

$$\mathfrak{n}_{\mathfrak{p}}^i = \begin{cases} r, & \text{if } |D_{\mathfrak{p}}^i| \geq r. \\ |D_{\mathfrak{p}}^i|, & \text{if } |D_{\mathfrak{p}}^i| < r. \end{cases}$$

   Now, for every $\mathfrak{p} \in d_i$ define $\mathfrak{d}_{\mathfrak{p}}^i = \{(\mathfrak{p}, s) \mid 1 \leq s \leq \mathfrak{n}_{\mathfrak{p}}^i\}$. Thereafter, define $\mathfrak{d}_i = \bigcup_{\mathfrak{p}} \mathfrak{d}_{\mathfrak{p}}^i$. Then, $\mathfrak{d} = \mathfrak{d}_0\mathfrak{d}_1\cdots$.

From $\mathfrak{d}_i$, the MFO-model $\mathfrak{m}_i = (\mathfrak{d}_i, \iota_i)$ can be extracted in the standard way. The full model is $\mathfrak{m} = \mathfrak{m}_0\mathfrak{m}_1 \cdots$

2. Show that $\mathfrak{m}, 0 \models \psi_0$. This will need the following definition and the two facts.

   Definition 6.3.7. *$\forall i \in \omega$, $\forall s$, $1 \leq s < r$, for every $\langle a_1, \cdots, a_s\rangle \in D_i^s$ and $\langle \mathfrak{a}_1, \cdots, \mathfrak{a}_s\rangle \in \mathfrak{d}_i^s$ such that $\langle a_1, \cdots, a_s\rangle$ **matches** $\langle \mathfrak{a}_1, \cdots, \mathfrak{a}_s\rangle$, if following conditions hold:*

   (a) $\forall s' : 1 \leq s' \leq s$, $\forall p \in P_\varphi$, $I_i(a_{s'}, p) = \top$ iff $\iota_i(\mathfrak{a}_{s'}, p) = \top$.

   (b) $\forall s', s'' : 1 \leq s', s'' \leq s$, $a_{s'} = a_{s''}$ iff $\mathfrak{a}_{s'} = \mathfrak{a}_{s''}$.

$fact_1$: $\forall i \in \omega$, $\forall s$, $1 \leq s < r$, for every $\langle a_1, \cdots, a_s\rangle \in D_i^s$ and $\langle \mathfrak{a}_1, \cdots, \mathfrak{a}_s\rangle \in \mathfrak{d}_i^s$ such that $\langle a_1, \cdots, a_s\rangle$ **matches** $\langle \mathfrak{a}_1, \cdots, \mathfrak{a}_s\rangle$, then for every $a_{s+1} \in D_i$, there exists an $\mathfrak{a}_{s+1} \in \mathfrak{d}_i$ such that $\langle a_1, \cdots, a_s, a_{s+1}\rangle$ **matches** $\langle \mathfrak{a}_1, \cdots, \mathfrak{a}_s, \mathfrak{a}_{s+1}\rangle$.

$fact_2$: $\forall i \in \omega$, $\forall s$, $1 \leq s < r$, for every $\langle a_1, \cdots, a_s\rangle \in D_i^s$ and $\langle \mathfrak{a}_1, \cdots, \mathfrak{a}_s\rangle \in \mathfrak{d}_i^s$ such that $\langle a_1, \cdots, a_s\rangle$ **matches** $\langle \mathfrak{a}_1, \cdots, \mathfrak{a}_s\rangle$, then for every $\mathfrak{a}_{s+1} \in \mathfrak{d}_i$, there exists an $a_{s+1} \in D_i$ such that $\langle a_1, \cdots, a_s, a_{s+1}\rangle$ **matches** $\langle \mathfrak{a}_1, \cdots, \mathfrak{a}_s, \mathfrak{a}_{s+1}\rangle$.

3. Define $CL(\Phi, \mathfrak{d}_i)$, for each $i \in \omega$.

4. Define $At^{\Phi,\mathfrak{d}}$, for each $i \in \omega$, as follows:

$$\forall \alpha \in CL(\Phi, \mathfrak{d}_i), \alpha \in At_i^{\Phi,\mathfrak{d}} \text{ iff } \mathfrak{m}, i \models \alpha$$

   With $At^{\Phi,\mathfrak{d}}$ in hand, we can compute $(\exists X)(At_i)$ and $(\forall X)(At_i)$ in a straightforward manner.

5. Define At_i^X as follows:

   (a) $At_i^X = \{x_j = x_j \mid 1 \leq j \leq r\}$.

   (b) $\forall j$, $2 \leq j \leq r$ there are two cases:

      i. $x_j \in (\forall X)(At_i)$, put $x_j = x_{j-1} \in At_i^X$.

ii. $x_j \in (\exists X)(At_i)$, if $\exists j' \leq j$ such that for some $\mathfrak{a} \in \mathfrak{d}_i$, $\{\beta_j[\mathfrak{a}/x_j], \beta_{j-1}[\mathfrak{a}/x_{j-1}]\} \subseteq At_i^{\Phi,\mathfrak{d}}$ then take the largest such j' and put $x_j = x_{j'}$. Else, put $x_j \neq x_{j'}$, for every $1 \leq j' \leq j$.

(c) for every j, j' if $x_j = x_{j'} \notin At_i^{\Phi,\mathfrak{d}}$, then put $x_j \neq x_{j'} \in At_i^X$.

(d) Take transitive closure of At_i^X.

6. Define $At_i = At_i^X \cup At_i^\Psi \cup At_i^{\Phi,\mathfrak{d}_i}$.

7. It is trivial to see that $At_i^X, At_i^\Psi, At_i^{\Phi,\mathfrak{d}_i}$ satisfy the required properties.

8. Define un_i's as follows: Fix $un_0 = \emptyset$. Then, for all $i \in \omega$, un_{i+1} is:

$$un_{i+1} = \begin{cases} \{\psi_1 \mathbf{U} \psi_2 \in At_{i+1}^\Psi \mid \psi_2 \notin At_{i+1}^\Psi\}, & \text{if } un = \emptyset. \\ \{\psi_1 \mathbf{U} \psi_2 \in un_i \mid \psi_2 \notin At_{i+1}^\Psi\}, & \text{otherwise.} \end{cases}$$

9. So, we have got the required sequence $\rho = (At_0, un_0)(At_1, un_1)(At_2, un_2)\cdots$. It is straighforward to check that ρ is a valid and accepting run.

□

Then, the following is immediate.

Theorem 6.3.8. *Given a $\mathcal{L}_{SPS}$-formula ψ_0 with $|\psi_0| = n$, the satisfiability of ψ_0 can be checked in time $2^{O(n \cdot r \cdot 2^k)}$, where r is the number of variable symbols occurring in ψ_0 and k is the number of predicate symbols occurring in ψ_0.*

Proof. Given ψ_0, ψ_0 is satisfiable iff $Lang(A_{\psi_0})$ is non-empty. $\because$ A_{ψ_0} is a Büchi automaton, the language emptiness of A_{ψ_0} can be checked in time $O(|Q|)$. The size of Q is computed as follows: $|Q| = |\mathbf{AT}| \cdot 2^{|UR|} = |\mathbf{AT}^X| \cdot |\mathbf{AT}^\Psi| \cdot |\mathbf{AT}^\Phi| \cdot 2^{|UR|}$. $|\mathbf{AT}|$ turns out to be $O(2^{n \cdot r \cdot 2^k})$ and $|UR| = |cl^\Psi| = O(2^n)$. Therefore, $|Q| = O(2^{n \cdot r \cdot 2^k})$. □

In order to specify SPS, in which clients do nothing but send a request of type u and wait for an answer, the most we can say about a client x is whether a request from x is pending or not. So the set of client properties are $P_c = \{req_u, ans_u \mid u \in \Gamma_0\}$. When $req_u(x)$ holds at some instant i, it means there is a pending request of type u from x at i. When $ans_u(x)$ holds at i, it means either there was no request from x or the request from x has already been answered. That is, $ans_u(x) \overset{\text{def}}{=} \neg req_u(x)$.

For the above sublogic, that is $\mathcal{L}_{SPS}$ with $P_c = \{req_u \mid u \in \Gamma_0\}$, we assert the following theorem, which can be inferred directly from Theorem 6.3.8.

Theorem 6.3.9. *Let ψ_0 be an $\mathcal{L}_{SPS}$ formula with $|Var(\psi_0)| = r$ and $|\Gamma_0| = k$ and $|\psi_0| = n$. Then, satisfiability of ψ_0 can be checked in time $O(2^{n \cdot r \cdot 2^k})$.*

6.4 Model Checking Problem for $\mathcal{L}_{SPS}$

For model checking the client-server system is modelled as an SPS, M, and the specification is given by a formula ψ_0 in $\mathcal{L}_{SPS}$. The problem is to check if the system M satisfies the specification ψ_0, denoted by $M \models \psi_0$. In order to do this we bound the SPS using ψ_0 and define an interpreted version.

Bounded Interpreted SPS

Let $M = (S, \delta, I, F)$ be an SPS and ψ_0 be a specification in $\mathcal{L}_{SPS}$. From ψ_0 we get $Var_u(\psi_0)$, for each $u \in \Gamma_0$. Now, let $\mathfrak{n} = (\Sigma_u r_u) \cdot k$ where $|\Gamma_0| = k$ and $|Var_u(\psi_0)| = r_u$. $\mathfrak{n}$ is the bound for SPS M. Now, for each $u \in \Gamma_0$ $CN_u = \{(i, u) \mid 1 \leq i \leq r_u, u \in \Gamma_0\}$ and $CN = \bigcup_u CN_u$. For each u, define $\mathbb{CN}_u = \{\{(j, u) \mid 1 \leq j \leq i\} \mid 1 \leq i \leq r_u\} \cup \{\emptyset\}$. Thereafter, define $\mathbb{CN} = \Pi_{u \in \Gamma_0} \mathbb{CN}_u$. Now, we have $\mathcal{CN} = \bigcup_{\mathbb{C} \in \mathbb{CN}} \mathbb{C}$. Now, we are in a position to define an interpreted form of bounded SPS. The interpreted SPS $\mathcal{M} = (\Omega, \Rightarrow, \mathcal{I}, \mathcal{F}, Val)$ is as follows:

1. $\Omega = S \times \mathcal{CN}$
2. $\mathcal{I} = \{(s, C) \mid s \in I, C = \emptyset\}$
3. $\mathcal{F} = \{(s, C) \mid s \in F, C = \emptyset\}$
4. $Val : \Omega \to (2^{P_s} \times \mathcal{CN})$
5. $\Rightarrow \subseteq \Omega \times \Gamma \times \Omega$ as follows: $(s, C) \xRightarrow{r} (s', C')$ iff $(s, r, s') \in \delta$ and the following conditions hold:

   (a) when $r = \tau$, $C = C'$.

(b) when $r = req_u$, $CN - C \neq \emptyset$, if $a \in CN_u - C$ is the least in the enumeration then $C' = C \cup \{a\}$.

(c) when $r = ans_u$, $X = \{a \in C \mid\} \cap CN_u \neq \emptyset$, $C' = C - \{a\}$ where $a \in X$ is the least in the enumeration.

Note, that, $|\mathcal{CN}| = \Pi_{u \in \Gamma_0}(r_u) < r^k$. Now, if, $|S| = l$, then $|\Omega| = O(l \cdot r^k)$.

We define the language of interpreted SPS $\mathcal{M}$ as follows

$$Lang(\mathcal{M}) = \{Val(c_0)Val(c_1)\cdots \mid c_0r_1c_1r_2c_2\cdots \text{ is a good run in } \mathcal{M}\}$$

We say that M satisfies ψ_0 if $Lang(\mathcal{M}) \subseteq A_{\psi_0}$, where A_{ψ_0} is the formula automaton of ψ_0. This holds when $Lang(\mathcal{M}) \cap Lang(A_{\neg\psi_0}) = \emptyset$. Therefore, the complexity to check emptiness of the product automaton, is linear in the product of the sizes of $\mathcal{M}$ and A_{ψ_0}.

Theorem 6.4.1. *$M \models \psi_0$ can be checked in time $O(l \cdot r^k \cdot 2^{n \cdot r \cdot 2^k})$.*

6.5 $\mathcal{L}_{SAS}$

In this section we describe a logical language to specify and verify SAS-like systems. Such a language has two mutually exclusive dimensions. One, captured by MFO fragment, talks about the plurality of clients. The other, captured by LTL fragment, talks about the temporal variations of server client interaction. Furthermore, the MFO fragment has to be multi-sorted to cover the multiplicity of service types. Keeping these issues in mind, we frame a logical language, which we call $\mathcal{L}_{SAS}$. Also, note that closing MFO sentences with temporal modalities, as in the case of $\mathcal{L}_{SPS}$, is not enough, since active clients are engaged with the server for a period in a non-trivial interaction. Therefore, we need to have free variables extending across temporal instances. But allowing more than one free variable in the scope of temporal modalities leads to undecidable logics. So, we describe an $MFOTL$ fragment with suitable constraints on the specifications in a way that they are expressive enough and have decidable reasoning algorithm too.

6.5.1 The Logic $\mathcal{L}_{SAS}$

The set of client formulae, Δ_u, for each type $u \in \Gamma_0$, is the modal closure of atomic client formulae P_c^u:

$$\alpha, \beta \in \Delta_u ::= p \mid \neg\alpha \mid \alpha \vee \beta \mid \Diamond\alpha$$

The set of server formulae, Ψ, is the modal closure of monodic formulae $\Phi = \{(\exists x : u)\alpha \mid \alpha \in \Delta_u, x \in Var_u, u \in \Gamma_0\}$.

$$\psi \in \Psi ::= q \mid \neg q \mid (\exists x : u)\alpha \in \Phi \mid \psi_1 \vee \psi_2 \mid \psi_1 \wedge \psi_2 \mid \Diamond\psi \mid \Box\psi$$

6.5.2 Semantics

The models of $\mathcal{L}_{SAS}$ are defined over a finite set of client specifications CS. A client specification of type u, $\mathfrak{a} \in CS$, is a finite sequence $\mathfrak{p}_0\mathfrak{p}_1 \cdots \mathfrak{p}_{\mathfrak{n}_\mathfrak{a}-1}$ where:

1. $\mathfrak{n}_\mathfrak{a}$ is the length of the sequence $\mathfrak{a}$ and
2. for all $0 \leq j < \mathfrak{n}_\mathfrak{a}$, $\mathfrak{p}_j \subset_{fin} P_c^u$.

For every model $M = (\nu, V, \xi)$ there is a map $\mathfrak{Z} : CN \times \mathbb{N}_0 \to CS'$ such that the client names, which are countable, can be mapped to the finite set CS. CS' contains client specification along with the local states, $CS' = \{(\mathfrak{a}, s) \mid \mathfrak{a} \in CS, 0 \leq s < \mathfrak{n}_\mathfrak{a}\}$.

Formally, a model is a triple $M = (\nu, V, \xi)$ where

1. $\nu = \nu_0\nu_1\nu_2 \cdots \nu_{len}$, where for all $0 \leq i \leq len$, $\nu_i \subset_{fin} P_s$,
2. $V = V_0V_1V_2 \cdots V_{len}$, where, for all $0 \leq i \leq len$, V_i is a finite subset of CN, gives the set of live agents at the ith instance.
   For every $0 \leq i < len$, V_i and V_{i+1} satisfy the following properties:
   (a) $V_i \subseteq V_{i+1}$ and for every $a \in V_{i+1} - V_i$ such that $\mathfrak{Z}(a, i+1) = (\mathfrak{a}, 0)$.
   (b) $V_{i+1} \subseteq V_i$ and for every $a \in V_i - V_{i+1}$ such that $\mathfrak{Z}(a, i) = (\mathfrak{a}, \mathfrak{n}_\mathfrak{a} - 1)$.
   Consequently, V satisfies the following interesting property. For every $a \in CN$, for every $i \in \omega$ if $a \in V_i$ then there exists $j > i$ such that $a \notin V_j$. Therefore, for every $a \in CN$ and $i \in \omega$ such that $a \in V_i$, we

define the left and right boundaries of the live window for a, denoted by $left(a,i)$ and $right(a,i)$, where $left(a,i) \leq i \leq right(a,i)$.

3. $\xi = \xi_0\xi_1\xi_2\cdots\xi_{len}$, where for all $0 \leq i \leq len$, $\xi_i : V_i \rightarrow 2^{P_c}$ gives the properties satisfied by each live agent at ith instance, in other words, the corresponding states of live agents. In terms of $\mathfrak{Z}$, $\xi_i(a) = \mathfrak{a}[s]$ where $\mathfrak{Z}(a,i) = (\mathfrak{a}, s)$ Alternatively, ξ_i can be given as $\xi_i : V_i \times P \rightarrow \{\top, \bot\}$, an equivalent form.

Let $\mathfrak{M}$ be the set of all $\mathcal{L}_{SAS}$ models. We define a subclass of $\mathcal{L}_{SAS}$ models, called $\mathfrak{RegM}$ as follows: An $M = (\nu, V, \xi) \in \mathfrak{RegM}$ if the following condition holds:

For every $0 \leq i \leq len$ and $\mathfrak{a} \in CS$, if there exists $a \in CN$ such that $\mathfrak{Z}(a,i) = (\mathfrak{a}, s)$ and $s > 0$ then for every other $b \neq a \in CN$ if $\mathfrak{Z}(b,i) = (\mathfrak{a}, s')$ then $s' = 0$.

That is, for every $\mathfrak{a} \in CS$, at most one instance of $\mathfrak{a}$ can be making moves at any point of time, while all other instances will be in wait in their initial state.

Satisfiability Relations $\models$, $\models_x$

Let $M = (\nu, V, \xi)$ be a valid model and $\pi : Var \times \omega \rightarrow CN$ be a partial map.

With legal M and π, the satisfying relations $\models$ and $\models_x$, for $x \in Var_u$, can be defined, via induction over the structure of $\psi \in \Psi$, and $\alpha \in \Delta_u$, respectively, as follows:

1. $M, i, \models q$ iff $q \in \nu_i$.

2. $M, i, \models \neg q$ iff $q \notin \nu_i$.

3. $M, i \models (\exists x : u)\alpha$ iff $\exists a \in CN_u : a \in V_i$ and $M, [x, i \mapsto a], i \models_x \alpha$.

4. $M, i \models \psi_1 \vee \psi_2$ iff $M, i \models \psi_1$ or $M, i \models \psi_2$.

5. $M, i \models \psi_1 \wedge \psi_2$ iff $M, i \models \psi_1$ and $M, i \models \psi_2$.

6. $M, i \models \Diamond\psi$ iff $\exists j \geq i$, $M, j \models \psi$.

7. $M, i \models \Box\psi$ iff $\forall j \geq i$, $M, j \models \psi$.

8. $M, [x, i \mapsto a], i \models_x p$ iff $\xi_i(a, p) = \top$.

9. $M, [x, i \mapsto a], i \models_x \neg\alpha$ iff $M, [x, i \mapsto a], i \not\models_x \alpha$.

10. $M, [x, i \mapsto a], i \models_x \alpha \vee \beta$ iff $M, [x, i \mapsto a], i \models_x \alpha$ or $M, [x, i \mapsto a], i \models_x \beta$.

11. $M, [x, i \mapsto a], i \models_x \Diamond\alpha$ iff $\exists j : i \leq j \leq right(\pi(x), i)$, $M, [x, i \mapsto a], j \models_x \alpha$.

6.5.3 Specification Examples

Even though $\mathcal{L}_{SAS}$ is a weak fragment of monadic monodic temporal logic, it is good enough to express some interesting properties of distributed systems with unbounded number of agents. In this section we describe a number of specifications written in $\mathcal{L}_{SAS}$ for the travel agency system.

The travel agency system has a single server, the travel agent tm with three types of clients $\Gamma_0 = \{h, a, t\}$, where h is for hotel accommodation, a is for airline reservation and t is for train reservation. The client activity alphabets are as follows: $\Theta_h = \{of, ro, co\}$, $\Theta_a = \{lf, df, os, nc\}$, $\Theta_t = \{cf, lt, ac, dt\}$ where the individual symbols have the following meaning: of off-season rates, ro rooms available, lf low fares, os on-flight service, cf confirmed ticket, ac A/C berths and dt direct train.

Here, we give specifications for each client (h, a and t) and the server (travel agent tm) of the composite travel agency service.

1. Specifications for Airline Reservation:

   (a) $(\Box\exists z : a)(nc(z))$
   There is at least one service offering no-hassles no-check entry

   (b) $\Box(\exists z : a)(\neg os(z) \supset df(z))$
   There is at least one offer for direct flights when there are no on-flight services

2. Specifications for Train Reservation:

   (a) $\Box(\exists y : t)(ac(y))$
   There is at least one service offering A/C tickets

   (b) $\Box(\exists y : t)((dt(y) \wedge cf(y)))$
   There exists at least one service offering confirmed ticket with no train changes

3. Specifications for Hotel Accommodation:

   (a) $\Box(\exists x : h)((of(x) \vee ro(x)))$
   There exists at least one hotel service offering off-season rates or cheap rooms

   (b) $\Box(\exists x : h)(co(x))$
   There is at least one hotel service offering high-end cottages

4. Specifications for Travel Agent:

   (a) $\Box((\exists x : h)(\Diamond of \vee \Diamond ro)(x) \wedge (\exists y : a)\Diamond lf(y) \vee (\exists z : t)\Diamond cf(z)))$
   the travel agent expects at least one hotel service offering off season rates or cheap rooms along with low fare flights offer from at least one airline service or confirmed berths from train service.

6.6 Satisfiability Problem for $\mathcal{L}_{SAS}$

In this section we show that the satisfiability problem of $\mathcal{L}_{SAS}$ is decidable. This is made possible by converting $\mathcal{L}_{SAS}$ formulae into multi-counter automata in the spirit of [84]. That is, given $\mathcal{L}_{SAS}$ formula ψ_0 we construct a multi-counter automaton A_{ψ_0} such that the following holds:

$$Lang(A_{\psi_0}) \text{ is non-empty if and only if } Models(\psi_0) \text{ is non-empty}$$

At the outset, we define subformula closure set cl from which we will construct state sets of the counter automaton A_{ψ_0}. cl is the smallest set satisfying the following conditions:

1. $\psi_0 \in cl$.

2. if $\neg q \in cl$ then $q \in cl$.

3. if $\psi_1 \vee \psi_2 \in cl$ then $\psi_1, \psi_2 \in cl$.

4. if $\psi_1 \wedge \psi_2 \in cl$ then $\psi_1, \psi_2 \in cl$.

5. if $\Diamond\psi \in cl$ then $\psi \in cl$.

6. if $\Box\psi \in cl$ then $\psi \in cl$.

Simultaneously, for each $u \in \Gamma_0$, we define CL_u and T_u as follows:

1. if $(\exists x : u)\alpha \in cl$ then $\alpha \in CL_u$ and $\alpha \in T_u$.
2. $\neg\alpha \in CL_u$ iff $\alpha \in CL_{u_j}$, $\neg\neg\alpha$ and α to be equivalent.
3. if $\alpha \vee \beta \in CL_u$ then $\alpha, \beta \in CL_u$.
4. if $\Diamond\alpha \in CL_u$ then $\alpha \in CL_u$.

Let $UR = \{\Diamond\psi \in cl\}$ be the set of all $\Diamond$ formulae in closure set cl. The individual closure sets for each $u \in \Gamma_0$ are computed as follows:

$$cl_u = CL_u \cup \{\neg\alpha \mid \alpha \in cl_u\}.$$

For each $u \in \Gamma_0$, let $Prop_u = \{p_{1,u}, \cdots, p_{\kappa_u,u}\}$ be the client propositions of type u occurring in ψ_0. Also, let Var_u be the variable symbols of type u occurring in ψ_0. Let $Var = \bigcup_{u\in\Gamma_0} Var_u = \{x_1, x_2, \cdots, x_{\mathfrak{n}_x}\}$. Let $(\exists X)(\psi_0) = \{(\exists x : u)\alpha \in cl\}$ be the set of all existential formulae occurring in ψ_0.

For each $u \in \Gamma_0$, T_u gives the client specifications of type u. An $\alpha \in T_u$ can be rewritten as disjunction of formulae of the type: $\beta_1 \wedge \Diamond(\beta_2 \wedge \Diamond(\beta_3 \wedge \cdots \Diamond\beta_t))$, where $\beta_1, \beta_2, \cdots, \beta_t$ are boolean and therefore can be represented by $\delta_1, \delta_2, \cdots, \delta_t$ such that for each $1 \leq j \leq t$, $\delta_j \subseteq Prop_u$. Let $\mathbb{D}_u$ be the set of all such client specifications of type u. Let $\mathbb{D} = \bigcup_{u\in\Gamma_0} \mathbb{D}_u$. Let $|\mathbb{D}_u| = \mathfrak{n}_j$ and $\mathfrak{n} = \Sigma_{1\leq j\leq k}\mathfrak{n}_j$. Also, suppose, there is a map $\mathfrak{m} : \mathbb{D} \to [\mathfrak{n}]$. For a particular $(\exists x : u)\alpha \in \exists X(\psi_0)$, $wit(\alpha) = \{\mathfrak{a} \in \mathbb{D}_u \mid \mathfrak{a}[0, \cdots, \mathfrak{n}_{\mathfrak{a}} - 1] \models \alpha$ in LTL sense$\}$, where $\mathfrak{n}_{\mathfrak{a}}$ is the length of $\mathfrak{a}$.

$A \subseteq cl$ is a counter automaton atom if following conditions hold:

1. for every $q \in cl$; $\{q, \neg q\} \not\subseteq A$;
2. for every $\psi_1 \vee \psi_2 \in cl$; if $\psi_1 \vee \psi_2 \in A$ then $\psi_1 \in A$ or $\psi_2 \in A$;
3. for every $\psi_1 \wedge \psi_2 \in cl$; if $\psi_1 \wedge \psi_2 \in A$ then $\psi_1 \in A$ and $\psi_2 \in A$;
4. for every $\Box\psi \in cl$; if $\Box\psi \in A$ then $\psi \in A$.

Let AT be the set of all such counter automaton atoms. Now, we can define counter automaton states, using AT, UR and $\mathbb{D}$. Suppose $D \subseteq \mathbb{D}$ is a set of client specifications. Let $\mathcal{D} = \{(\mathfrak{a}, s) \mid \mathfrak{a} \in D, 0 \leq s < \mathfrak{n}_{\mathfrak{a}}\}$. $\mathfrak{d} \subseteq \mathcal{D}$ is a valid client set template if following condition holds:

For each $\mathfrak{a} \in D$ if $(\mathfrak{a}, s), (\mathfrak{a}, s') \in \mathfrak{d}$ then either $s = s'$ or one of the s ors' is 0.

Let $\mathfrak{D}$ be the set of all such ($\mathfrak{d}$) subsets. The set of states is $Q \subseteq AT \times 2^{UR} \times \mathfrak{D}$ where the following condition holds:

for every $(\exists x : u)\alpha \in cl$ if $(\exists x : u)\alpha \in A$ then $\exists \mathfrak{a} \in wit(\alpha)$ such that $(\mathfrak{a}, 0) \in \mathfrak{d}$.

A single counter automaton state $q \in Q$ is a triple $\langle A, \mathfrak{u}, \mathfrak{d} \rangle$. A is an atom of the kind described above. $\mathfrak{d} \in \mathfrak{D}$ is a *representative* set of all **active client specifications** and their individual local states.

1. $\mathfrak{d}$ contains objects of the kind $(\mathfrak{a}, s)$.
2. $\mathfrak{d}$ contain at most two entries of $\mathfrak{a}$.
3. $\mathfrak{d}$ helps crucially in extracting a model from a run.

A counter automaton configuration is a pair $\langle q, \widetilde{n} \rangle$ where,

1. $q \in Q$ and
2. $\widetilde{n}$ is a tuple with a counter value $\widetilde{n}[\mathfrak{a}]$, for each $\mathfrak{a} \in \mathbb{D}$.

A state $(At, \mathfrak{u}, \mathfrak{d})$ is initial if $\psi_0 \in A$, $u = \emptyset$. A state $(At, \mathfrak{u}, \mathfrak{d})$ is final if $u = \emptyset$, $\mathfrak{d} = \emptyset$.

We discover that the existential fragment of $\mathcal{L}_{SAS}$ can be decided by a heavy dose of non-determinism in the formula (counter) automaton. We are helped by the fact that every good run in the formula (counter) automaton ends in a final state with $\mathfrak{d} = \emptyset$ and every counter, for every $\mathfrak{a} \in \mathbb{D}$, $\widetilde{n}[\mathfrak{a}]$ with value 0. This will make sure that every active copy of every active client in the formula automaton moves in a fair manner.

The transitions of the counter automaton are labelled by the set $L = \Pi_{\mathfrak{a} \in \mathbb{D}} L_{\mathfrak{a}}$, where for each $\mathfrak{a} \in \mathbb{D}$, $L_{\mathfrak{a}} = \{\mathbf{dec}, \tau, \mathbf{nz}, \mathbf{inc}\}$. Formally, the transition relation $\rightarrow \subseteq Q \times L \times Q$ is defined as follows: $\langle At, \mathfrak{u}, \mathfrak{d}_1 \rangle \xrightarrow{\tau} \langle B, \mathfrak{v}, \mathfrak{d}_2 \rangle$ if following conditions hold:

1. for every $\Diamond\psi \in cl$ if $\psi \in B$ then $\Diamond\psi \in A$.

2. for every $\Box\psi \in cl$ if $\Box\psi \in A$ then $\Box\psi \in B$.

3. $$\mathfrak{v} = \begin{cases} \{\Diamond\psi \in B \mid \psi \notin B\}, & \text{if } \mathfrak{u} = \emptyset. \\ \{\Diamond\psi \in u \mid \psi \notin B\}, & \text{otherwise.} \end{cases}$$

4. For every $1 \leq j \leq \mathfrak{n}_x$, for every $(\exists x_j : u)\alpha_j \in cl$ the following conditions hold:

   (a) if $(\exists x_j : u)\alpha_j \notin A$ and $(\exists x_j : u)\alpha_j \in B$ then for at least one $\mathfrak{a} \in wit(\alpha)$, $(\mathfrak{a}^j, 0) \in \mathfrak{d}_2$ and $r[\mathfrak{a}^j] = \textbf{inc}$.

   (b) if $(\exists x_j : u)\alpha_j \in A$ and $(\exists x_j : u)\alpha_j \in B$ then for at least one $\mathfrak{a} \in wit(\alpha)$, $(\mathfrak{a}^j, 0) \in \mathfrak{d}_2 \cap \mathfrak{d}_1$ and $r[\mathfrak{a}^j] = \tau$.

   (c) if $(\exists x_j : u)\alpha_j \in A$ and $(\exists x_j : u)\alpha_j \notin B$ or $(\exists x_j : u)\alpha_j \notin A$ and $(\exists x_j : u)\alpha_j \notin B$ then for at least one $\mathfrak{a} \in wit(\alpha)$ one of the following conditions holds:

      i. if $\{(\mathfrak{a}^j, 0)\} \subseteq \mathfrak{d}_1$ then either $\{(\mathfrak{a}^j, 0)\} \subseteq \mathfrak{d}_2$ or $\{(\mathfrak{a}^j, 0), (\mathfrak{a}^j, 1)\} \subseteq \mathfrak{d}_2$ and $r[\mathfrak{a}^j] = \textbf{nz}$.

      ii. if $\{(\mathfrak{a}^j, s),\} \subseteq \mathfrak{d}_1$ then $\{(\mathfrak{a}^j, s+1)\} \subseteq \mathfrak{d}_2$ and $r[\mathfrak{a}^j] = \tau$.

      iii. if $\{(\mathfrak{a}^j, 0), (\mathfrak{a}^j, s),\} \subseteq \mathfrak{d}_1$ then $\{(\mathfrak{a}^j, 0), (\mathfrak{a}^j, s{+}1)\} \subseteq \mathfrak{d}_2$ and $r[\mathfrak{a}^j] = \tau$.

      iv. if $\{(\mathfrak{a}^j, 0), (\mathfrak{a}^j, \mathfrak{n}_\mathfrak{a} - 1),\} \subseteq \mathfrak{d}_1$ then $\{(\mathfrak{a}^j, 0)\} \subseteq \mathfrak{d}_2$ and $r[\mathfrak{a}^j] = \textbf{dec}$.

      v. if $\{(\mathfrak{a}^j, \mathfrak{n}_\mathfrak{a} - 1),\} \subseteq \mathfrak{d}_1$ then $\{\} \subseteq \mathfrak{d}_2$ and $r[\mathfrak{a}^j] = \textbf{dec}$.

Let $\rho = (q_0, \widetilde{n}_0)\overset{r_1}{\rightarrow}(q_1, \widetilde{n}_1)\overset{r_2}{\rightarrow}(q_2, \widetilde{n}_2)\overset{r_3}{\rightarrow}\cdots$ be an accepting run of A_{ψ_0}. Extraction of a valid model from a run ρ crucially depends on the fact that the run ends with all counters $n_\mathfrak{a} = 0$ and the additional set containing client specifications with local states, $\mathfrak{d}$ being empty. Let $M = (\nu, V, \xi)$ be the $\mathcal{L}_{SAS}$-model extracted from ρ. M is defines as follows:

1. $\nu = \nu_0\nu_1\nu_2\cdots$ where for every $i \in \omega$, $\nu_i = A_i \cap P_s$;

2. We define $\zeta = \zeta_0\zeta_1\cdots$ and unfold it to $\xi = \xi_0\xi_1\cdots$. Simultaneously we define $V = V_0V_1V_2\cdots$ too.

(a) V_0 is obtained from $\mathfrak{d}_0$. $\mathfrak{d}_0$ is divided in $\mathfrak{d}_0^1 \cdots \mathfrak{d}_0^{\mathfrak{n}_x}$. For every $(\exists x_j)\alpha_j \in A_0$, $\mathfrak{d}_0^j = \{(\mathfrak{a}^j, 0)\}$. For every $(\exists x_{j'})\alpha_{j'} \notin A_0$, $\mathfrak{d}_0^{j'} = \emptyset$.

Let $V_0 = \{[\mathfrak{a}^j, 0] \mid (\mathfrak{a}^j, 0) \in \mathfrak{d}_0\}$. $[\mathfrak{a}^j, 0]$ is the **identity** of the client of type $\mathfrak{a}$ introduced at the 0th instance as a witness to the existential formula $(\exists x_j : u)\alpha_j$.

ζ_0 is defined as follows: for every $a = [\mathfrak{a}^j, 0] \in V_0$, $\zeta_0([\mathfrak{a}^j, 0]) = \mathfrak{a}[0]$.

(b) Suppose we have defined $V_0 V_1 V_2 \cdots V_i$ and $\zeta_0\zeta_1\zeta_2 \cdots \zeta_i$. Looking at $(q_i, \widetilde{n}_i) \overset{r_{i+1}}{\rightarrow} (q_{i+1}, \widetilde{n}_{i+1})$ we define V_{i+1} and ζ_{i+1} as follows: For each $1 \leq j \leq \mathfrak{n}_x$,

$(\exists x_j)\alpha_j \in A_i - A_{i-1}$: In this case $r[\mathfrak{a}^j] = 1$. Therefore, we add a new client name to the system. Hence, $[\mathfrak{a}^j, i] \in V_i$. $[\mathfrak{a}^j, i]$ is the **identity** of the client of type $\mathfrak{a}$ introduced at the ith instance as a witness to the existential formula $(\exists x_j : u)\alpha_j$.

Also, $\zeta_i([\mathfrak{a}^j, i]) = \mathfrak{a}_j[0]$. Every other client of the type $\mathfrak{a}^j$ remains as it is, i.e., for all $[\mathfrak{a}^j, i''] \in V_{i-1}$ such that $i'' < i$ we have $[\mathfrak{a}^j, i''] \in V_i$ and $\zeta_i([\mathfrak{a}^j, i'']) = \zeta_{i-1}([\mathfrak{a}^j, i''])$.

$(\exists x_j)\alpha_j \in A_{i-1} \cap A_i$: In this case take the client $[\mathfrak{a}^j, i']$ with the largest $0 \leq i' < i$ in V_{i-1} and $\zeta_i([\mathfrak{a}^j, i']) = \mathfrak{a}[0]$. Every other client of the type $\mathfrak{a}^j$ remains as it is, i.e., for all $[\mathfrak{a}^j, i''] \in V_{i-1}$ such that $i'' < i'$ we have $[\mathfrak{a}^j, i''] \in V_i$ and $\zeta_i([\mathfrak{a}^j, i'']) = \zeta_{i-1}([\mathfrak{a}^j, i''])$.

otherwise:) Let $[\mathfrak{a}^j, i'] \in V_{i-1}$ be the client with the smallest $0 \leq i' < i$. Let $\zeta_{i-1}([\mathfrak{a}^j, i']) = \mathfrak{a}[s]$. We consider two mutually exclusive cases.

$0 < s \leq \mathfrak{n}_\mathfrak{a} - 1$: In this case, we have $(\mathfrak{a}^j, s) \in \mathfrak{d}_{i-1}$. If $r_i[\mathfrak{a}^j] = \tau$ then $[\mathfrak{a}^j, i'] \in V_i$ and $\zeta_i([\mathfrak{a}^j, i']) = \mathfrak{a}[s+1]$. If $r_i[\mathfrak{a}^j] = -1$ then $[\mathfrak{a}^j, i'] \notin V_i$. Every other client of the type $\mathfrak{a}^j$ remains as it is, i.e., for all $[\mathfrak{a}^j, i''] \in V_{i-1}$ such that $i'' > i'$ we have $[\mathfrak{a}^j, i''] \in V_i$ and $\zeta_i([\mathfrak{a}^j, i'']) = \zeta_{i-1}([\mathfrak{a}^j, i''])$.

$s = 0$: In this case, we are not sure to have $(\mathfrak{a}^j, 0) \in \mathfrak{d}_{i-1}$. But, we are assured by the following fact which holds otherwise ρ won't be accepting. $[\mathfrak{a}^j, i']$ will get meaning from $i^\dagger$ onwards.

Fact 6.6.1. *$\exists i^\dagger \geq i$ such that $(\mathfrak{a}^j, 0) \in \mathfrak{d}_{i^\dagger - 1}$ and $r_{i^\dagger}[\mathfrak{a}^j] = \mathbf{nz}$.*

Proof. $[\mathfrak{a}^j, i']$ was added at i'th instance, after executing an $r_{i'}$

with $r_{i'}[\mathfrak{a}^j] = \mathbf{inc}$. Clearly, this means, $|\{[\mathfrak{a}^j, i''] \in V_i \mid i'' \leq i\}| = \widetilde{n}_i[\mathfrak{a}^j]$. Because, ρ ends with $\widetilde{n}[\mathfrak{a}^j] = 0$, therefore, we need to have a corresponding $i^\ddagger$ instance which executes $r_{i^\ddagger}$ with $r_{i^\ddagger}[\mathfrak{a}^j] = \mathbf{dec}$. The transition with $r_{i^\ddagger}[\mathfrak{a}^j] = \mathbf{dec}$ takes place along with $\mathfrak{d}^j_{i^\ddagger-1} = \{(\mathfrak{a}^j, \mathfrak{n}_\mathfrak{a} - 1)\}$. From the definition of $\rightarrow$, we can go back in the run ρ and find an $i^\dagger < i^\ddagger$ where $\mathfrak{d}^j_{i^\dagger-1} = \{(\mathfrak{a}^j, 0)\}$ and $r_{i^\dagger}[\mathfrak{a}^j] = \mathbf{nz}$. □

Lemma 6.6.2. *For every $i \in \omega$, for every $\psi \in cl$, if $\psi \in A_i$ then $M, i \models \psi$.*

Proof. We prove the truth lemma by induction on the structure of ψ. The propositional and modal cases are straightforward. We look at the monodic case.

$\psi \equiv (\exists x : u)\alpha$**:** Let $(\exists x : u)\alpha \in A_i$. We have to show $M, i \models (\exists x : u)\alpha$.

When $i = 0$, $(\exists x : u)\alpha \in A_0$. By initial state conditions, $\exists \mathfrak{a} \in \mathbb{D}_u$ such that $(\mathfrak{a}, 0) \in \mathfrak{d}_0$ and $\alpha \in \mathfrak{a}[0]$. By another, simple induction, we can show, in the LTL sense, $\mathfrak{p}^\mathfrak{a}_0 \cdots \mathfrak{p}^\mathfrak{a}_{\mathfrak{n}_\mathfrak{a}} \models \alpha$. By the definition of M, there exists $a \in CN$, such that $M, [x \mapsto a], 0 \models_x \alpha$. Which means, $M, 0 \models (\exists x)\alpha$ and we are done.

When $i = len$, $(\exists x)\alpha \notin A_{len}$, otherwise, ρ won't be accepting.

Now, suppose, $0 < i < len$. There are two possibilities here.

$(\exists x : u)\alpha \notin A_{i-1}$**:** There is a witness to α in $\mathfrak{d}_{i-1}$. By another induction hypothesis, $(\exists x)\alpha \notin A_{i-1}$ implies $M, i-1 \models (\exists x : u)\alpha$.

$(\exists x)\alpha \in A_{i-1}$**:** By the definition of M, there exists $[\mathfrak{a}, i] \in V_i$ such that $\alpha \in \mathfrak{a}[0]$. We have also defined $\zeta_i([\mathfrak{a}, i]) = \mathfrak{a}[0]$ and $\xi([\mathfrak{a}, i]) = \mathfrak{a}[0] \cap P_c$. By the definition of M, following fact holds:

Fact 6.6.3. *$\exists i' > i$, $[\mathfrak{a}, i] \notin V_{i'}$ and $\forall i'' : i \leq i'' < i'$, $[\mathfrak{a}, i] \in V_{i''}$. Furthermore, $\zeta_i([\mathfrak{a}, i]) \cdots \zeta_{i'}([\mathfrak{a}, i])$ is a stuttered form of $\mathfrak{a}$.*

Using the above fact, we can now show, via another simple induction that $M, [x \mapsto [\mathfrak{a}, i]], i \models_x \alpha$, which in turn means that $M, i \models (\exists x : u)\alpha$, and we are done.

□

Given $M = (\nu, V, \xi)$ of ψ_0 we want to describe a good run ρ of A_{ψ_0}. First, we observe the following:

Lemma 6.6.4. *Given a model* $M = (\nu, V, \xi) \in \mathfrak{M}$ *there is an* $M' = (\nu', V', \xi') \in \mathfrak{RegM}$ *such that* $M, 0 \models \psi_0$ *iff* $M', 0 \models \psi_0$.

Proof. Let len be the length of the model M. Given M, we divide $\{0, 1, \cdots, len\}$ into three sets as follows:

- For every $(\exists x_j)\alpha_j \in (\exists X)(\psi_0)$ we find those instances $0 \leq i < len$ where a new client of a type in $wit(\alpha_j)$ is introduced in V_i of M by assertion of $(\exists x_j)\alpha_j$ at i. Let $FrshIns_j$ be the set of all such points, computed as follows:

  $FrshIns_j = \{0 < i < len \mid M, i \models (\exists x_j)\alpha_j$ but $M, i-1 \not\models (\exists x_j)\alpha_j\} \cup \{0 \mid M, i \models (\exists x_j)\alpha_j\}$.

- Let $OldIns_j$ be the set of all points, where an old client is asserted again. This set is computed as follows:

  $OldIns_j = \{0 < i < len \mid M, i \models (\exists x_j)\alpha_j$ also $M, i-1 \models (\exists x_j)\alpha_j\}$.

- $NoFrshIns_j$ is the set of all points where $(\exists x_j)\alpha_j$ is not asserted, i.e., $NoFrshIns_j = \{0 \leq i \leq len \mid M, i \not\models (\exists x_j)\alpha_j\}$ which is, essentially, $\{0 \leq i \leq len\} - (FrshIns_j \cup OldIns_j)$.

Let $FrshIns_j = \{i^j_1, \cdots, i^j_{fpnts_j}\}$ be all these fresh points. For a particular point i^j_s, $1 \leq s \leq fpnts_j$,

- let $FrshClts^{i_s}_j = \{a^{ijs}_1, \cdots, a^{ijs}_{r_{ijs}}\}$ be the active client witnesses introduced at i^j_s and
- $FrshClTypes^{i_s}_j = \{\mathfrak{a}^{ijs}_1, \cdots, \mathfrak{a}^{ijs}_{r_{ijs}}\}$ be their respective types.

Now, we give an algorithm to construct M' using the above information. We describe the algorithm in brief. Due to the inherent property of $\mathcal{L}_{SAS}$ it suffices to have at most one active instance of a client of a particular type at any point of time in the model of a given formula. So, we have gathered together the information about all client types introduced at all possible points in the model. We modify the model M in such a way that one instance of a particular type is duly introduced in M' whenever a similar one is introduced in M. As regards the activity of clients in the modified model at most one client of a particular type, introduced by jth existential formula, makes a move and only at those instances which are in $NoFrshIns_j$. The particular instance which moves is decided by the natural order induced by order on $\mathbb{D}$ and $FrshIns_j$. Due to this constraint, the length of the modified model M' would be larger than that of M.

1. $V'_0 = \{[\mathfrak{a}, (j, 0)] \mid \mathfrak{a} \in FrshClTypes^0_j\}$.

   $\zeta'_0 : V'_0 \to \mathcal{D}$ such that for each $[\mathfrak{a}, (j, 0)] \in V'_0$, $\zeta'_0([\mathfrak{a}, (j, 0)]) = (\mathfrak{a}, 0)$.

2. Inductively, if V'_{i-1} and ζ'_{i-1} are defined then we want to define V'_i and ζ'_i using the information collected above.

   For each $1 \leq j \leq \mathfrak{n}_x$, we compute the clients which are being introduced afresh by the assertion of $(\exists x_j : u_j)\alpha_j$ at i. This set is as follows:

   $NewCltIns^i_j = \{[\mathfrak{a}, (j, i)] \mid \mathfrak{a} \in FrshClTypes^i_j\}$ and could be empty. Thereafter, we define $NewCltIns^i = \bigcup\limits_j NewCltIns^i_j$, the corresponding set over all existential formulae. Also, we compute those clients which exit the system as ith instance.

   $ExClIns^i = \{[\mathfrak{a}, (j, i')] \in V'_{i-1} \mid \zeta'_{i-1}([\mathfrak{a}, (j, i')]) = (\mathfrak{a}, \mathfrak{n}_\mathfrak{a}-1), i \in NoFrshClts^i_j\}$. Now, V'_i turns out to be $V'_{i-1} - ExClIns^i \cup NewClIns^i$.

   To compute ζ'_i we define more sets.

   - Let $\mathbb{J}_i = \{1 \leq j \leq \mathfrak{n}_x \mid [\mathfrak{a}, (j, i')] \in ExClIns^j\}$ be all those j's such that a client introduced by $(\exists x_j : u_j)\alpha_j$ earlier moves out in the present instance.
   - We compute all those clients, apart from those which are exiting, whose state change,

     $ModClIns^i = \{a^{j,i^\dagger}_\mathfrak{a} \in V'_{i-1} \mid j \notin \mathbb{J}_i, i^\dagger \text{ is least in order for } j\}$.
   - $RestClIns^j = V'_i - ModClIns^i$ are all those clients which remain waiting at the initial state.

   For all $[\mathfrak{a}, (j, i')] \in ModClIns^i$ if $\zeta'_{i-1}([\mathfrak{a}, (j, i')]) = (\mathfrak{a}, s)$ then $\zeta'_i([\mathfrak{a}, (j, i')]) = (\mathfrak{a}, s + 1)$.

   For all $[\mathfrak{a}, (j, i')] \in RestClIns^i$ if $\zeta'_i([\mathfrak{a}, (j, i')]) = (\mathfrak{a}, 0)$.

ξ is obtained from ζ in the standard way. Note, that, the way M' is constructed it satisfies the following:

Fact 6.6.5. *For each $1 \leq j \leq \mathfrak{n}_x$, For each $0 \leq i \leq len$ if $i \in FrshIns_j$ then For every $a \in FrshClts_i$ of type $\mathfrak{a} \in wit(\alpha_j)$ there exists at least one $a^i_j \in V'_i$ of type $\mathfrak{a}$.*

Using the above fact, we can easily verify the following by induction on the structure of α:

Claim 6.6.6. *if $\exists a \in FrshClts_j$ and $M, [x_j \mapsto a], i \models_{x_j} \alpha$ then $\exists a^i_j \in V'_i$ such that $M, [x_j \mapsto a^i_j], i \models_{x_j} \alpha$.*

For all i, $0 \leq i \leq len$, $\nu'_i = \nu_i$ and for all i, $len \leq i \leq len'$, $\nu'_i = \nu_{len}$. Now, it is easy to verify the following claim:

Claim 6.6.7. *For all $1 \leq i \leq len$, for every $\psi \in subf(\psi_0)$, if $M, i \models \psi$ then $M', i \models \psi$.*

Proof. This claim is proved by induction on the structure of ψ. The propositional and modal cases are straightforward. So, we look at the monodic case.

$(\exists x_j : u_j)\alpha_j :)$ $M, i \models (\exists x_j)\alpha_j$

implies $\exists a \in V_i$ $M, [x_j \mapsto a], i \models_{x_j} \alpha_j$

Let us consider two cases:

$i \in FrshIns_j$**:** By the construction of M', $M', [x_j \mapsto a^i_j], i \models_{x_j} \alpha_j$. Therefore, we are done.

$i \in OldIns_j$**:** Let $i^\dagger < i$ be the latest instance in $FrshIns_j$. By the construction of M' we have for all $i'' : i^\dagger \leq i'' \leq i$, $M', [x_j \mapsto a^{i^\dagger}_j], i'' \models_{x_j} \alpha_j$. Therefore, we are done.

□

□

Assume that M already has the desired property. That is, $M \in \mathfrak{RegM}$. Now, we get down to the task of constructing the corresponding run ρ. For each $i \geq 0$, q_i is a triple $(At_i, u_i, \mathfrak{d}_i)$. We describe each element of the ith triple as follows:

For every $i \in \omega$, $A_i = \{\psi \in CL \mid M, i \models \psi\}$.

$u_0 = \emptyset$ and for all $i \geq 0$ u_{i+1} is defined as follows:

$$u_{i+1} = \begin{cases} \{\Diamond\psi \in A_{i+1} \mid \psi \notin A_{i+1}\}, & \text{if } u_i = \emptyset. \\ \{\Diamond\psi \in u_i \mid \psi \notin A_{i+1}\}, & \text{otherwise.} \end{cases}$$

Let $\mathbb{D} = \{\mathfrak{a} \in CS \mid \mathfrak{a} \in wit(\alpha_j), \text{ for some } 1 \leq j \leq k\}$. To construct $\mathfrak{d}_i$, for each $i \geq 0$, we use the extra machinery described already. For all $0 \leq i < len$, $\mathfrak{d}_i = \{(\mathfrak{a}, s) \mid \mathfrak{a} \in \mathbb{D} \text{ and } \exists a \in CN, \mathfrak{z}(a, i) = (\mathfrak{a}, s)\}$. Also, $\mathfrak{d}_{len} = \emptyset$.

We compute,

1. for all $1 \leq i \leq len$, r_i as follows:

   (a) For all $\mathfrak{b} \in CS$ such that there is no entry of $\mathfrak{b}$ in $\mathfrak{d}_{i-1}$ or $\mathfrak{d}_i$, then, $r_i[\mathfrak{b}] = \tau$,

   (b) For all $\mathfrak{a} \in CS$ such that there is an entry of $\mathfrak{a}$ in $\mathfrak{d}_{i-1}$ or $\mathfrak{d}_i$,

      i. if $(\mathfrak{a}, \mathfrak{n}_{\mathfrak{a}} - 1) \in \mathfrak{d}_{i-1} - \mathfrak{d}_i$ then $r_i[\mathfrak{a}] = \mathbf{dec}$,

      ii. if $\mathfrak{a} \in wit(\alpha_j)$ and $(\exists x_j)\alpha_j \in A_i - A_{i-1}$ then $r_i[\mathfrak{a}] = \mathbf{inc}$,

      iii. in all other cases $r_i[\mathfrak{a}] = \tau$.

2. for all $0 \leq i \leq len$, $\widetilde{n}_i$ as follows: We first compute $\widetilde{n}_0$ and then inductively define $\widetilde{n}_i$ when $\widetilde{n}_{i-1}$ is given.

   For all $\mathfrak{a} \in \mathbb{D}$ such that there is an entry of $\mathfrak{a}$ in $\mathfrak{d}_0$, $\widetilde{n}_0[\mathfrak{a}] = 1$. For all $\mathfrak{a} \in \mathbb{D}$ such that there is no entry of $\mathfrak{a}$ in $\mathfrak{d}_0$, $\widetilde{n}_0[\mathfrak{a}] = 0$.

   Once we know r_i computing $\widetilde{n}_i$ is trivial. For $\mathfrak{a} \in \mathbb{D}$,

   (a) if $r_i[\mathfrak{a}] = \tau$ then $\widetilde{n}_i[\mathfrak{a}] = \widetilde{n}_{i-1}[\mathfrak{a}]$,

   (b) if $r_i[\mathfrak{a}] = \mathbf{dec}$ then $\widetilde{n}_i[\mathfrak{a}] = \widetilde{n}_{i-1}[\mathfrak{a}] - 1$,

   (c) if $r_i[\mathfrak{a}] = \mathbf{inc}$ then $\widetilde{n}_i[\mathfrak{a}] = \widetilde{n}_{i-1}[\mathfrak{a}] + 1$.

It is easy to verify the following:

1. For all $0 \leq i \leq len$ $(At_i, u_i, \mathfrak{d}_i)$ is a valid state in Q,

2. For all $0 \leq i \leq len$ $\langle (At_i, u_i, \mathfrak{d}_i), \widetilde{n}_i \rangle$ is a valid configuration,

3. For all $0 \leq i < len$ $(At_i, u_i, \mathfrak{d}_i) \overset{r_{i+1}}{\rightarrow} (At_{i+1}, u_{i+1}, \mathfrak{d}_{i+1})$ is a legal transition, and $r \in L$.

4. $\langle (At_0, u_0, \mathfrak{d}_0), \widetilde{n}_0 \rangle$ is an initial configuration and $\langle (At_{len}, u_{len}, \mathfrak{d}_{len}), \widetilde{n}_{len} \rangle$ is a final configuration.

Thus, given an $\mathcal{L}_{SAS}$ formula ψ_0, we constructed a multi-counter automaton A_{ψ_0} and showed the following: $Lang(A_{\psi_0})$ is non-empty if and only if ψ_0 is satisfiable. As language emptiness of multi-counter automaton is decidable (non-elementary) we too have a non-elementary algorithm to check satisfiability of $\mathcal{L}_{SAS}$ formulae.

Observe that our logic does not have subformulae of the kind $(\forall x : u)\alpha$ in Φ. Even though the language remains monodic and therefore decidable [45]. In the presence of universal subformulae, we can no longer check it using multi-counter automata (without zero-check). This is due to the following reason: $(\exists x : u)\alpha$ translates to the increment of the counter corresponding to a witness $\mathfrak{a}$ of the client formula α which is allowed, but $(\forall x : u)\alpha$ translates to the zero-check of the counters corresponding to every witness $\mathfrak{a}$ of the client formula α which is clearly not allowed. Obviously, we need a stronger formalism to decide the satisfiability of logic $\mathcal{L}_{SAS}$ with $(\forall x : u)\alpha$ in Φ.

7

Discussion

We summarize the work done in the thesis below.

- In Chapter 2 we presented a partial order based scheme to model behaviours of SCMS systems. These models, called Lamport diagrams, were inspired from timing diagrams [60] and were discussed in [67] and [66]. In order to specify the appropriate set of behaviours of SCMS we proposed a local logic called wm-LTL, an extension of the logic m-LTL. The formulas of wm-LTL are interpreted over Lamport diagrams. This logic contained an immediate past modality, $\ominus_j$, originally from m-LTL of [67], a standard $\diamondminus$ modality, and a novel concurrent present modality, $\langle now \rangle_j$ modality, first discussed in [75] in the context of synchronous systems. We also discussed the suitability of wm-LTL for writing specifications of SCMS systems, namely Travel Agency Web service and Quote-request Web service. We also observed that this logic is bisimular invariant, with respect to n-agents Lamport diagrams as well as n-agent and $(n + m)$-agent Lamport diagrams, where the extra m agents are transparent to the logic and are introduced to implement the message passing channels between the n agents.

- In Chapter 3 we discussed an automaton model for SCMS systems, called Sequence of n Communicating Automata, for a given fixed n. SCAs are a variant of standard CFSM [16] and were first presented in [67] and [66]. m-LTL was shown to be decidable in [67]. Using the same technique, an extension of automata theoretic technique of [84], we showed that w-LTL has also decidable satisfiability and model checking properties. This crucially depended on the fact that models of *wm*-LTL formulas, Lamport diagrams, always have one bounded linearizations.

- In Chapter 4 we presented automaton models for the two type of SSMC under study, SPS for discrete systems and SAS for session oriented systems. We show that these models are equivalent to multi-counter automata and therefore, reachability and, consequently, language emptiness in SPS and SAS, are decidable. Furthermore, they are closed under union and intersection but not complementation.

- In Chapter 5 we observed that $MFOTL$ is an ideal candidate to specify SPS and SAS like client-server systems but is undecidable. Undecidability of $MFOTL$ was proved by encoding recurrent tiling problem [39]. This follows the treatment in [45].

- In Chapter 6 we presented two fragments of $MFOTL$ as possible candidates for specifying SPS and SAS, $\mathcal{L}_{SPS}$ and $\mathcal{L}_{SAS}$ respectively. We also showed their suitability with specification examples. $\mathcal{L}_{SPS}$ is known to have decidable properties, though we present a novel automata based algorithm to decide satisfiability and model checking. $\mathcal{L}_{SAS}$ is a fragment of monadic monodic logic [45] which, too, is known to be decidable. We have presented a novel multi-counter automata based scheme to decide the satisfiability of $\mathcal{L}_{SAS}$.

Further Work

While we presented the model of SCAs to describe SCMS systems, it must be noted that this is done only in terms of a convenient structure for obtaining a decision procedure. Modelling client-server systems at the right level of abstraction is an interesting challenge and "compiling" such models into automata requires a great deal of work. We hope that SCAs present a step in that direction. Many automata theoretic questions on SCAs also remain to be answered, notably that of complementation.

We described a local temporal logic to specify SCMS systems, called *wm*-LTL. *wm*-LTL is composed of two sublogics *m*-LTL [67] and a fresh one *w*-LTL. The expressiveness of these logics is an interesting exercise. Even though *wm*-LTL is embedded in a two-variable fragment of first order logic (FO) over Lamport diagrams, it is yet a very weak fragment. Note that the two variable fragment of FO over Lamport diagrams is undecidable so some weakening is indeed needed. We would like to explore and find out the FO equivalent of wm-LTL as well as those of individual sublogics. We expect that some type of guarded fragment [7] of FO may suffice. Also, what modalities can be added to wm-LTL retaining decidability is an interesting question.

The important theme unaddressed here is proof theory: how do we reason about client-server systems in these logics, in terms of proof principles? We hope that the formal aspects presented here will be of initial help in this direction.

The thesis contained two automata based models for client-server systems with unbounded number of clients and one server. It was also shown that these models were equivalent to multi-counter automata, therefore closed under union and intersection but not complementation. We would like to extend these models to client-server systems with multiple servers. That is, automaton models for composite systems which are a cross between SCMS and SSMC systems. Combining techniques that work on formalisms such as message sequence charts ([6]) with these models seems difficult.

Note that $\mathcal{L}_{SAS}$ does not have $\neg(\exists x : u)\alpha$ in the set of server formulas. As already pointed out in Chapter 6, only the existential fragment can be decided using multi-counter automaton. We would like to find out what kind of stronger automata models could be used to reason about the full fragment, or whether there are none.

As regards extended SPS/SAS models for multiple server and unbounded clients, we would like to have specification languages for such systems, preferably fragments of monadic monodic logic which is already known to be decidable.

An orthogonal exercise could be development of tools to efficiently implement the model checking problem for the system SPS/SAS against $\mathcal{L}_{SPS}/\mathcal{L}_{SAS}$ specifications, *á lá* MONA [42][54] or SPIN [47],[78].

Publications

1. S. Sheerazuddin, "Automata Models for Services with Unboundedly Many Clients", In International Conference on Software and Computing Technology (ICSCT), 2010.

2. S. Sheerazuddin, "Temporal Specifications for Services with Unboundedly Many Passive Clients", In International Conference on Distributed Computing and Networking, (ICDCN) 2011.

3. R. Ramanujam and S. Sheerazuddin, "Temporal Specifications for Services with Unboundedly Many Active Clients", In International Conference on Distributed Computing and Internet Technology (ICDCIT), 2011.

Bibliography

[1] *Distributed Systems: Methods and Tools for Specification, An Advanced Course, April 3-12, 1984 and April 16-25, 1985 Munich*, volume 190 of *Lecture Notes in Computer Science*. Springer, 1985.

[2] Parosh Aziz Abdulla and Bengt Jonsson. Verifying programs with unreliable channels. *Inf. Comput.*, 127(2):91–101, 1996.

[3] Parosh Aziz Abdulla and Bengt Jonsson. Verifying networks of timed processes (extended abstract). In *TACAS*, pages 298–312, 1998.

[4] Bowen Alpern and Fred B. Schneider. Defining liveness. *Inf. Process. Lett.*, 21(4):181–185, 1985.

[5] Rajeev Alur, Costas Courcoubetis, and David L. Dill. Model-checking for real-time systems. In *LICS*, pages 414–425, 1990.

[6] Rajeev Alur, Kousha Etessami, and Mihalis Yannakakis. Realizability and verification of MSC graphs. In *ICALP*, pages 797–808, 2001.

[7] Hajnal Andréka, Johan van Benthem, and István Németi. Back and forth between modal logic and classical logic. *Logic Journal of the IGPL*, 3(5):685–720, 1995.

[8] Krzysztof R. Apt and Dexter Kozen. Limits for automatic verification of finite-state concurrent systems. *Inf. Process. Lett.*, 22(6):307–309, 1986.

[9] Mohamed Faouzi Atig, Ahmed Bouajjani, and Shaz Qadeer. Context-bounded analysis for concurrent programs with dynamic creation of threads. In *TACAS*, pages 107–123, 2009.

[10] Francesco Belardinelli and Alessio Lomuscio. A complete quantified epistemic logic for reasoning about message passing systems. In *CLIMA VIII*, pages 248–267, 2007.

[11] Francesco Belardinelli and Alessio Lomuscio. A quantified epistemic logic for reasoning about multiagent systems. In *AAMAS*, page 87, 2007.

[12] Francesco Belardinelli and Alessio Lomuscio. A complete first-order logic of knowledge and time. In *KR*, pages 705–714, 2008.

[13] Francesco Belardinelli and Alessio Lomuscio. Interactions between time and knowledge in a first-order logic for multi-agent systems. In *KR*, 2010.

[14] M. Ben-Ari. *Principles of concurrent and distributed programming*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1990.

[15] Ahmed Bouajjani, Javier Esparza, and Oded Maler. Reachability analysis of pushdown automata: Application to model-checking. In *CONCUR*, pages 135–150, 1997.

[16] Daniel Brand and Pitro Zafiropulo. On communicating finite-state machines. *J. ACM*, 30(2):323–342, 1983.

[17] Michael C. Browne, Edmund M. Clarke, and Orna Grumberg. Reasoning about networks with many identical finite state processes. *Inf. Comput.*, 81(1):13–31, 1989.

[18] J. Richard Büchi. On a decision method in restricted second order arithmetic. *Z. Math. Logik Grundlag Math*, (6):66–92, 1960.

[19] Jerry R. Burch, Edmund M. Clarke, Kenneth L. McMillan, David L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Inf. Comput.*, 98(2):142–170, 1992.

[20] Olaf Burkart and Bernhard Steffen. Composition, decomposition and model checking of pushdown processes. *Nord. J. Comput.*, 2(2):89–125, 1995.

[21] Karlis Cerans. Deciding properties of integral relational automata. In *ICALP*, pages 35–46, 1994.

[22] Vassilis Christophides, Richard Hull, Gregory Karvounarakis, Akhil Kumar, Geliang Tong, and Ming Xiong. Beyond discrete e-services: Composing session-oriented services in telecommunications. In *TES*, pages 58–73, 2001.

[23] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs*, pages 52–71, 1981.

[24] Edmund M. Clarke, E. Allen Emerson, and A. Prasad Sistla. Automatic verification of finite state concurrent systems using temporal logic specifications: A practical approach. In *POPL*, pages 117–126, 1983.

[25] Edmund M. Clarke, Orna Grumberg, and Michael C. Browne. Reasoning about networks with many identical finite-state processes. In *PODC*, pages 240–248, 1986.

[26] Edmund M. Clarke, Orna Grumberg, and Somesh Jha. Verifying parameterized networks using abstraction and regular languages. In *CONCUR*, pages 395–407, 1995.

[27] Edmund M. Clarke, Orna Grumberg, and David E. Long. Model checking and abstraction. In *POPL*, pages 342–354, 1992.

[28] Edmund M. Clarke, Orna Grumberg, and Doron Peled. *Model Checking.* MIT Press, 2000.

[29] Anatoli Degtyarev, Michael Fisher, and Alexei Lisitsa. Equality and monodic first-order temporal logic. *Studia Logica*, 72(2):147–156, 2002.

[30] E. Allen Emerson and Vineet Kahlon. Parameterized model checking of ring-based message passing systems. In *CSL*, pages 325–339, 2004.

[31] E. Allen Emerson and Kedar S. Namjoshi. Reasoning about rings. In *POPL*, pages 85–94, 1995.

[32] E. Allen Emerson and Kedar S. Namjoshi. Automatic verification of parameterized synchronous systems (extended abstract). In *Proceedings of the 8th International Conference on Computer Aided Verification*, CAV '96, pages 87–98, London, UK, 1996. Springer-Verlag.

[33] Javier Esparza. Petri nets, commutative context-free grammars, and basic parallel processes. In *FCT*, pages 221–232, 1995.

[34] Dov M. Gabbay, Ian Hodkinson, and Mark Reynolds. *Temporal Logic: Mathematical Foundations and Computational Aspects Volume 1.* Clarendon Press, 1994.

[35] Dov M. Gabbay, Ian M. Hodkinson, and Mark A. Reynolds. *Temporal Logic. Part 1.* Clarendon Press, 1994.

[36] Steven M. German and A. Prasad Sistla. Reasoning about systems with many processes. *J. ACM*, 39(3):675–735, 1992.

[37] Erich Grädel. On the restraining power of guards. *J. Symb. Log.*, 64(4):1719–1742, 1999.

[38] Joseph Y. Halpern and Moshe Y. Vardi. The complexity of reasoning about knowledge and time. I. Lower Bounds. *J. Comput. Syst. Sci.*, 38(1):195–237, 1989.

[39] David Harel. Recurring dominoes: Making the highly undecidable highly understandable (preliminary report). In *FCT*, pages 177–194, 1983.

[40] David Harel. Effective transformations on infinite trees, with applications to high undecidability, dominoes, and fairness. *J. ACM*, 33(1):224–248, 1986.

[41] Matthew Hennessy and Robin Milner. Algebraic laws for nondeterminism and concurrency. *J. ACM*, 32(1):137–161, 1985.

[42] Jesper G. Henriksen, Jakob L. Jensen, Michael E. Jørgensen, Nils Klarlund, Robert Paige, Theis Rauhe, and Anders Sandholm. Mona: Monadic second-order logic in practice. In *TACAS*, pages 89–110, 1995.

[43] Ian M. Hodkinson. Monodic packed fragment with equality is decidable. *Studia Logica*, 72(2):185–197, 2002.

[44] Ian M. Hodkinson, Roman Kontchakov, Agi Kurucz, Frank Wolter, and Michael Zakharyaschev. On the computational complexity of decidable fragments of first-order linear temporal logics. In *TIME*, pages 91–98, 2003.

[45] Ian M. Hodkinson, Frank Wolter, and Michael Zakharyaschev. Decidable fragment of first-order temporal logics. *Ann. Pure Appl. Logic*, 106(1-3):85–134, 2000.

[46] Ian M. Hodkinson, Frank Wolter, and Michael Zakharyaschev. Decidable and undecidable fragments of first-order branching temporal logics. In *LICS*, pages 393–402, 2002.

[47] Gerard J. Holzmann. The model checker spin. *IEEE Trans. Software Eng.*, 23(5):279–295, 1997.

[48] John E. Hopcroft and Jean-Jacques Pansiot. On the reachability problem for 5-dimensional vector addition systems. *Theor. Comput. Sci.*, 8:135–159, 1979.

[49] Michaela Huhn, Peter Niebert, and Frank Wallner. Model checking logics for communicating sequential agents. In *FoSSaCS*, pages 227–242, 1999.

[50] Walter Hussak. Decidable cases of first-order temporal logic with functions. *Studia Logica*, 88(2):247–261, 2008.

[51] Petr Jancar. Decidability of a temporal logic problem for petri nets. *Theor. Comput. Sci.*, 74(1):71–93, 1990.

[52] Yonit Kesten, Oded Maler, Monica Marcus, Amir Pnueli, and Elad Shahar. Symbolic model checking with rich assertional languages. *Theor. Comput. Sci.*, 256(1-2):93–112, 2001.

[53] Yonit Kesten, Amir Pnueli, Elad Shahar, and Lenore D. Zuck. Network invariants in action. In *CONCUR*, pages 101–115, 2002.

[54] Nils Klarlund, Anders Møller, and Michael I. Schwartzbach. Mona implementation secrets. In *CIAA*, pages 182–194, 2000.

[55] S. Rao Kosaraju. Decidability of reachability in vector addition systems (preliminary version). In *STOC*, pages 267–281, 1982.

[56] Robert P. Kurshan and Kenneth L. McMillan. A structural induction theorem for processes. In *PODC*, pages 239–247, 1989.

[57] Robert P. Kurshan and Kenneth L. McMillan. A structural induction theorem for processes. *Inf. Comput.*, 117(1):1–11, 1995.

[58] Jean-Luc Lambert. A structure to decide reachability in petri nets. *Theor. Comput. Sci.*, 99(1):79–104, 1992.

[59] Leslie Lamport. Proving the correctness of multiprocess programs. *IEEE Trans. Software Eng.*, 3(2):125–143, 1977.

[60] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.

[61] Leslie Lamport and Nancy A. Lynch. Distributed computing: Models and methods. In *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics (B)*, pages 1157–1199. 1990.

[62] Kamal Lodaya and Ramaswamy Ramanujam. Tense logics for local reasoning in distributed systems. In *FSTTCS*, pages 71–88, 1991.

[63] Kamal Lodaya, Ramaswamy Ramanujam, and P. S. Thiagarajan. Temporal logics for communicating sequential agents: I. *Int. J. Found. Comput. Sci.*, 3(2):117–159, 1992.

[64] Kamal Lodaya and P. S. Thiagarajan. Decidability of a partial order based temporal logic. In *ICALP*, pages 582–592, 1993.

[65] Ernst W. Mayr. An algorithm for the general petri net reachability problem. In *STOC*, pages 238–246, 1981.

[66] B. Meenakshi. *Reasoning About Distributed Message Passing Systems.* PhD thesis, 2004.

[67] B. Meenakshi and Ramaswamy Ramanujam. Reasoning about message passing in finite state environments. In *ICALP*, pages 487–498, 2000.

[68] R. Milner. *Communication and concurrency.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989.

[69] Marvin L. Minsky. *Computation: finite and infinite machines.* Prentice-Hall, Upper Saddle River, NJ, USA, 1967.

[70] Carlos Olarte and Frank D. Valencia. Undecidability of monadic first-order linear-time temporal logic. *Studia Logica*, 82:1–13, 2006.

[71] David Michael Ritchie Park. Concurrency and automata on infinite sequences. In *Theoretical Computer Science*, pages 167–183, 1981.

[72] Amir Pnueli. The temporal logic of programs. In *FOCS*, pages 46–57, 1977.

[73] Amir Pnueli, Jessie Xu, and Lenore D. Zuck. Liveness with (0, 1, infty)-counter abstraction. In *CAV*, pages 107–122, 2002.

[74] Jean-Pierre Queille and Joseph Sifakis. Specification and verification of concurrent systems in cesar. In *Symposium on Programming*, pages 337–351, 1982.

[75] Ramaswamy Ramanujam. Locally linear time temporal logic. In *LICS*, pages 118–127, 1996.

[76] John H. Reif and A. Prasad Sistla. A multiprocess network logic with temporal and spatial modalities. In *ICALP*, pages 629–639, 1983.

[77] E. Rudolph, P. Graubmann, and J. Grabowski. Tutorial on message sequence charts. 28, 1996.

[78] Theo C. Ruys and Gerard J. Holzmann. Advanced spin tutorial. In *SPIN*, pages 304–305, 2004.

[79] Ze'ev Shtadler and Orna Grumberg. Network grammars, communication behaviors and automatic verification. In *Automatic Verification Methods for Finite State Systems*, pages 151–165, 1989.

[80] A. Prasad Sistla and Steven M. German. Reasoning with many processes. In *LICS*, pages 138–152, 1987.

[81] Robert Endre Tarjan. Depth-first search and linear graph algorithms. *SIAM J. Comput.*, 1(2):146–160, 1972.

[82] Salvatore La Torre, P. Madhusudan, and Gennaro Parlato. Model-checking parameterized concurrent programs using linear interfaces. In *CAV*, pages 629–644, 2010.

[83] Johan van Benthem. *Exploring logical dynamics.* Center for the Study of Language and Information, Stanford, CA, USA, 1997.

[84] Moshe Y. Vardi and Pierre Wolper. An automata-theoretic approach to automatic program verification (preliminary report). In *LICS*, pages 332–344, 1986.

[85] Pierre Wolper. Expressing interesting properties of programs in propositional temporal logic. In *POPL*, pages 184–193, 1986.

[86] Pierre Wolper and Bernard Boigelot. Verifying systems with infinite but regular state spaces. In *CAV*, pages 88–97, 1998.

[87] Pierre Wolper and Vinciane Lovinfosse. Verifying properties of large sets of processes with network invariants. In *Automatic Verification Methods for Finite State Systems*, pages 68–80, 1989.

[88] Frank Wolter and Michael Zakharyaschev. Axiomatizing the monodic fragment of first-order temporal logic. *Ann. Pure Appl. Logic*, 118(1-2):133–145, 2002.

[89] Lenore D. Zuck and Amir Pnueli. Model checking and abstraction to the aid of parameterized systems (a survey). *Computer Languages, Systems & Structures*, 30(3-4):139–169, 2004.