Centre Fédéré en Vérification

Technical Report number 2004.35

The Computational Complexity of Scenario-based Agent Verification and Design

Yves Bontemps, Pierre-Yves Schobbens

This work was partially supported by a FRFC grant: 2.4530.02

http://www.ulb.ac.be/di/ssd/cfv

The Computational Complexity of Scenario-based Agent Verification and Design

Yves Bontemps [a,1] Pierre-Yves Schobbens [a]

[a] *University of Namur, Institut d'Informatique*

Abstract

We first advocate that the AUML (Agent Unified Modelling Language) notation, even in its new version, is not precise enough to adequately describe protocols. This problem was long identified by Harel and we propose to follow his solution: extend sequence diagrams with a "prechart", i.e. single out the initiation sequence of the protocol. This new notation keeps readability and intuition, but is also technically adequate and is given a formal semantics. It actually is a form of simple temporal logics, equipped with a game-based semantics, which is appropriate for the modeling of agent-based systems. We then go on to study its complexity. Unsurprisingly, the version with protocol roles is undecidable. The main interesting problem is to synthesize agents that follow the protocol described. Surprisingly, it is undecidable even if we remove roles, alternatives, loops, asynchronous communication, conditions, constraints, negations (already removed in AUML). The complexity of checking whether a society of agents obeys a protocol given in this trivial notation is also surprisingly high: it is in PSPACE-complete, like temporal logic, while we show that this simple language is strongly less expressive than temporal logic. Notations in-between have the expected increase in expressiveness, but no increase in complexity. This justifies the use of a language including alternatives, asynchronous communication and conditions, since it increases expressiveness with no cost in complexity.

Key words: Scenarios; Strategies; AUML; Live Sequence Charts; Synthesis; Model Checking; Computational Complexity.

1 Introduction

Agents are autonomous entities that react to changes in their environment, according to defined plans. Their behavior follows these plans, which are mo-

[1] FNRS Research Fellow

tivated by goals. In order to achieve their goals, which are realized through plans, agents need to coordinate. This coordination has to follow well-specified agent coordination protocols.

There are two possible approaches to ensure that these two constraints are met. The first possibility is to *verify* that a certain agent description complies with the description of the protocols and goals. The second possibility is to check that agents *can be designed* to follow the protocols. The second approach is clearly more ambitious, as it proposes to automate the construction of agents design models.

In this paper, we consider the promising scenario-based approach for specifying protocols. Scenario-based graphical languages are widely used, in many different forms, for illustrating and specifying protocols [1]. Message Sequence Charts, which are standardized by the International Telecommunication Union, are by far the most popular of these languages [2]. They present, in an intuitive way, how processes interact, through message passing. This language has been incorporated in the UML, as "Interaction Diagrams" [3]. In the agent world, FIPA is defining a unified language, based on UML, called Agent Unified Modeling Language (AUML) [4], for modeling agent systems. However, this language also inherits the problems that are found both in UML and in ITU languages. First, UML 2.0 only partially specifies the semantics of Interaction Diagrams, which opens the way to ambiguities [5]. Second, MSCs carry much implicit information. In particular, engineers draw the same diagrams with different intents: sometimes, they just want to describe *some* trace of a protocol, sometimes, they intend to describe *all possible* reactions to a certain message or protocol initiation sequence. However, these different meanings are implicit: there are no syntactic constructs carrying this information. For this reason, Damm and Harel have introduced Live Sequence Charts [6]. This language extends exactly MSCs (and Interaction Diagrams) with those syntactic constructs. Hence, one can distinguish between provisional and mandatory behavior.

Actually, Live Sequence Charts provide engineers with a graphical front-end to Temporal Logic [7,8]. However, this language remains (i) graphical and (ii) scenario-based. In [9], we have shown that LSCs can be smoothly equipped with a game-based semantics, hence making it usable for agent systems specifications. We will thus use this language as a basis for verification and design of agents. Since this language is actually a form of Temporal Logics, these two problems are well-defined, in terms of classical logical problems. Agent verification is often called *model checking* [10], whereas agent design is dubbed *synthesis*.

Here, we show that many simple problems on (non-hierarchical) LSC have a surprisingly high complexity, and in particular that the automated synthesis

of a distributed agent system is undecidable. On afterthought, it is hardly surprising that distributed development is undecidable. This means that more knowledge has to be put in the synthesis algorithms, e.g. as heuristics [11], so as to alleviate the work of agent programmers in well-known cases, leaving them the more creative parts.

The work of Wooldridge and colleagues is related to what we present here [12,13]. They study the computational complexity of agent verification and agent design, with respect to task descriptions. A task description is represented as a subset of all runs, i.e. a language, which are acceptable. The complexity of verifying whether an agent design satisfies the task description, in a given environment, is described, as a function of the complexity (i.e. the complexity class to which belongs the language) of the task description. Crudely, their result is that, for task descriptions in Σ_u^p (i.e. recognizable in polynomial time by a Turing Machine with u calls to an NP oracle), the complexity of verification is Π_{u+1}^p, i.e. exactly one universal alternation is introduced. When the task description is PSPACE-complete, verification is PSPACE-complete as well.

Walton presents a lightweight language for describing agent dialogues, named Multi-Agent Protocol [14]. This language is based on the theory of Speech Act and is intended to be an alternative to Statecharts [15], which are used in Electronic Institutions [16]. Walton proposes a translation of MAP to PROMELA, the input language of the SPIN model-checker [17], which allows one to check MAP models against LTL formulae. Their work is more pragmatic than ours, but could be coupled with our approach. Here, we propose to use a graphical, user-friendly, language for specifying protocols and remain purposedly abstract on the actual form of agents implementing these protocols. MAP could be such an implementation language (although designed as a specification language). Another possibility would be to use agent-oriented programming languages, such as AgentSpeak [18], 3APL [19], ConGoLog [20], for instance. There is also some tool support for the verification of AgentSpeak programs [21]. First, Agent Speak programs are made finite, then they are translated to PROMELA. Bordini *et al.* also present a logic based on BDI (Beliefs-Desires-Intentions) for specifying the requirements that Agent Speak programs should fulfill. These requirements are translated to LTL. Again, our scenario-based language could be used as a requirement language.

Wooldridge *et al.* present another language for agent programming, called MABLE, which is based on classical imperative languages, enriched with features from agent-oriented programming paradigm [22]. Essentially, it is possible to use a belief-desire-intention logic instead of classical boolean expressions. `if-then-else` constructs are modified into `if-then-else-unsure` constructs, to cope with the problem of agents not believing whether the condition holds true or false. It supports a form of inter-agent communication, in which

agents can inform or request information, through message passing, telling other agents about their mental state.

This feature will drive the reader to notice that LSCs and AUML, for describing agent protocols, are not as rich as FIPA's ACL [23]. With ACL, agents can communicate with other agents about their beliefs, desires and intentions, and require information about these facts as well. MABLE has been extended to support the verification that MABLE programs comply with a protocol description given in ACL [24]. These extensions entail an unsurprising high complexity. The goal of our paper is to show that even a very basic language will have a high complexity, we have thus purposedly excluded such features.

Such basic LSCs stemmed from the world of telecommunication. Quite naturally, they will find their way to the agent world, as well, as demonstrated by the presence of Interaction Diagrams in AUML.

The paper is structured as follows. We present, in Sec. 2.1, the syntax and semantics of Live Sequence Charts (LSC), that is used to specify the future system. We compare this language with the current AUML Interaction Diagrams and show that LSCs cope with the various ambiguities of AUML Interaction Diagrams. Agent models are given using an agent-oriented (i.e. distributed) state-based formalism, here input/output automata, encoding strategies, as presented in Sec. 2.2. This section concludes by defining when a design model is a correct implementation of a scenario-based specification. In Sec. 3, verification problems are considered. First, checking whether a design model is a correct implementation (Sec. 3.1) and then, whether a specification refines another specification (Sec. 3.2). The question of whether a specification is implementable is investigated in Sec. 4. Sec. 5 presents various constructs that can be added to our version of LSCs, making the language more expressive, but preserving all the results of this paper. Finally, in Sec. 6, we summarize the results and put them in perspective.

2 Models

We assume that we are given a finite set of *agent names* Ag and of message names $\mathcal{M}$. An event is a triple from $Ag \times \mathcal{M} \times Ag$. The set of events is Σ. We will denote the set of events "sent", or triggered, (resp. "received", or sensed) by some agent a with Σ_a^s (resp. Σ_a^r) and let $\Sigma_a = \Sigma_a^s \cup \Sigma_a^r$. An event of the form (a_1, m, a_2) represents the fact that a_1 sends message m to a_2. Σ^* represents the set of all finite sequences of events, while Σ^ω are all infinite sequences. We let $\Sigma^\infty = \Sigma^* \cup \Sigma^\omega$. For $\Sigma' \subseteq \Sigma$, *projection* $(w|_{\Sigma'})$ is the operation that removes from w all symbols that are not in Σ'.

We assume here, for simplicity, that communication is instantaneous. (In contrast, some undecidability proofs of [25] require the more complex FIFO communication). From agents' behavior emerge sequences of events, which we can observe. Hence, we identify behaviors and sequences of events.

2.1 Live Sequence Charts

Live Sequence Charts [6] are based on Message Sequence Charts (MSCs) [2]. They present the various interactions of agents. Every agent owns a "life-line", labeled by its name, e.g. "ui", "cm", "client1" in Fig. 4. Interactions take place through events, that are shown as arrows. An occurrence of (a_1, e, a_2) is displayed as an arrow labeled by m, from a_1's life-line to a_2's life-line. MSCs are unclear with respect to the "status" of a scenario, i.e. whether a scenario represents all possible behaviors or just some of them. They are also silent about the role of messages that do not appear in a scenario, viz. whether they are forbidden by their mere absence or whether they can appear at will. We call this feature *message abstraction*. Furthermore, engineers informally assign different status to messages: some of them activate, or trigger, the described scenario, whereas other are expected answers.

For instance, Fig. 1 presents an example of an interaction diagram. It is an excerpt of Misty Nodine's proposal of a solution to a FIPA case study for assessing Interaction Diagrams [26]. This case study is concerned with the modeling of the voting protocol followed by the United Nation Security Council for issuing resolutions. When presented with the scenario Fig. 1, it is unclear whether it states that "whenever a meeting is called, all members are called for a vote by the chair" or if it is a possible execution that has been singled out.

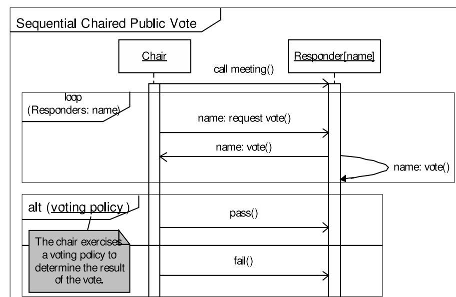


Fig. 1. Interaction Diagram (UN Vote Procedure)

LSCs clarify this [6]. They add syntactic constructs to MSCs to state explicitly whether the diagram is a mere example (existential scenarios) or constrains all behaviors of the future system (universal scenarios). The former are simply MSCs, surrounded by a dashed-line box. The latter are MSCs, divided

in two parts: an upper part, named *prechart*, that is graphically surrounded by an hexagonal dashed-line box, and a lower-part named *main chart* is the lower-part, surrounded by a solid-line rectangle. The intuitive semantics is “whenever the agents behave as in the prechart, they shall behave according to the main chart afterwards”. LSCs add “message abstraction” by explicitly stating which events are *restricted*. All events appearing in the LSC are automatically restricted. Additional events can be restricted thanks to a “restricts” clause. This provides the scenario with a scope (alphabet).

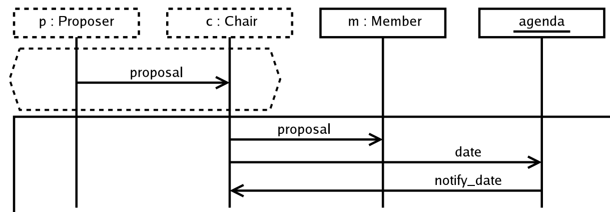


Fig. 2. Symbolic LSC (UN Proposal Scenario)

Harel and Marelly have extended LSC with symbolic instances [27]. This construct allows one to talk about the *roles* played by agents in protocols. The basic idea is to introduce first-order variables, that are placeholders for agents. These variables may be quantified, thus telling whether a scenario is applicable to *all* agents playing a certain role or to *one of them*. This is akin to universal/existential quantification, in logics. For instance, in Fig. 2, the scenario states that “if *some* proposer sends a proposal to some chair, this chair forwards this proposal to *all* members and decides of a date at which the vote will take place.” Of course, the voting date will eventually occur, which is the reason why the agenda notifies the chair. Thus, universal quantification is graphically denoted by inscribing variable names within solid-line boxes, whereas existential quantification corresponds to dashed-line boxes. It is also possible to refer to particular agents by their name, e.g. agenda. This is represented by underlining their name.

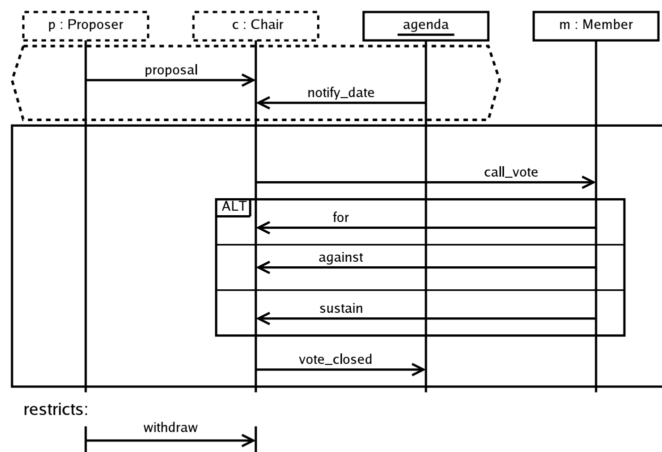


Fig. 3. Symbolic LSC (UN Voting Scenario)

Symbolic LSCs is a rich and powerful notation. However, when introducing quantification and unbounded agent populations, most analysis problems get undecidable. As an example, satisfiability is undecidable. We postpone the proof of this fact until section 5.2.

Actually, we obtain, in principle, a graphical version of first-order logic. In this paper, we mainly focus on a simpler version of LSCs that does not include roles. Thus, we will only take into account LSCs describing protocols for an *a priori* determined finite number of agents. Remark that, in the case of the UN protocol, the number of agents is actually finite, bounded and known beforehand: there are 15 members, among which 5 permanent members. Including roles in Interaction Diagrams provides engineers with a shorthand to avoid writing lengthy scenarios, but is not really necessary here.

Like Interaction Diagrams, the semantics of LSCs is based on a partial order. The temporal ordering of events is deduced from three constraints and their transitive closure: (1) life-lines induce a total ordering on their events, from top to bottom, (2) agents synchronize on shared events, i.e. two locations linked by an arrow are order-equivalent and (3) all locations in the prechart appear before main chart locations. In Message Sequence Charts (MSC) parlance, the prechart and main chart are *strongly sequenced*. For example, combining the clauses, in Fig. 4, events “getdata” and “updating” are unordered. Clause (1) can be relaxed thanks to *co-regions*. A co-region is a sequence of locations, belonging to the same life-line, along which a dashed line is drawn, see the two “getnew” events in Fig. 4.

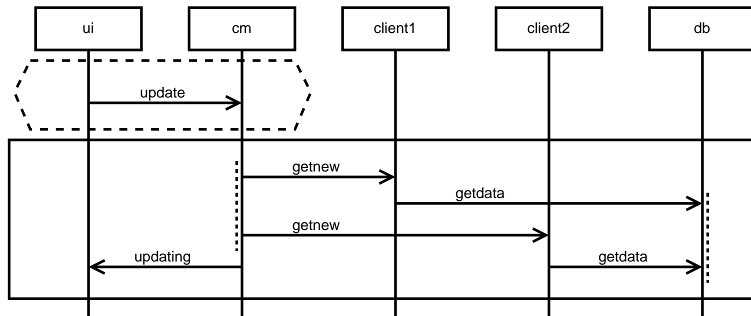


Fig. 4. Update Scenario (CTAS)

Live Sequence Charts have been used to model various real-life systems such as the weather synchronization logic of NASA’s Center TRACON Automation System (CTAS) [28], a radio-based train system [29], virtual wrappers for PCI bus [30] and some part of the C elegans worm [31]. Examples displayed in Fig. 4, 5, 6 and 7 are based on the CTAS system. This system aims at synchronizing various clients that make use of weather data reports. When new data is available, a certain protocol is followed to update all clients data. However, if some client fails to fetch the new report, the system tries to roll back to the previous version. The rationale is that all clients should always be

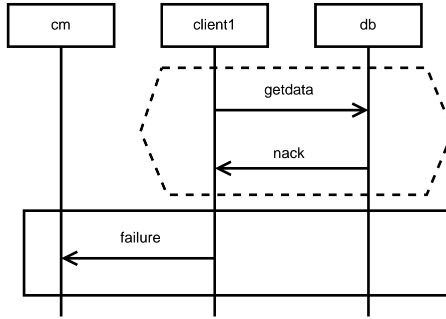


Fig. 5. Propagate Answers (CTAS)

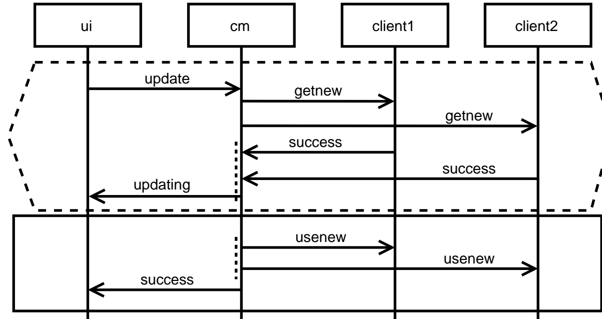


Fig. 6. Success Scenarios (CTAS)

using the same data. LSCs describe the following requirements:

Fig. 4: when the user asks for an update, all clients are asked to fetch the new weather reports. The user is notified of the updating process.

Fig. 5: whenever the database refuses a download, the cm (communication manager) is notified.

Fig. 6: if all clients report success, then they are confirmed that they should use the new data. The user is informed of the success.

Fig. 7: if some client fails to update its state, all clients are required to roll back to the previous state, *after* the user has been notified that the updating process is taking place.

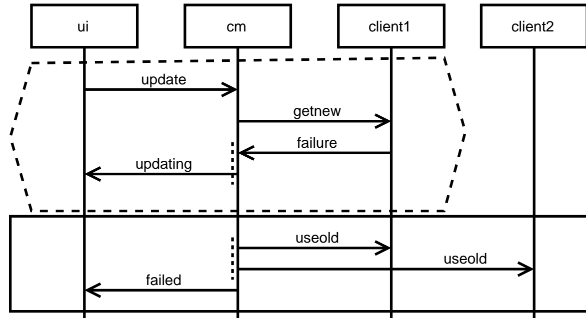


Fig. 7. Failure Scenarios (CTAS)

We now define formally the abstract syntax and the semantics of universal LSCs. Following the tradition of logics, this semantics is given through the

notions of interpretation and model.

Definition 1 (Labeled partial order (LPO)) *A Σ'-labeled partial order (LPO) is a tuple $\langle L, \leq, \lambda, \Sigma' \rangle$, where*

- *L is a set of* locations*. If L is finite, the LPO is called* finite*.*
- *$\leq \subseteq L \times L$ is a partial order on L (a transitive, anti-symmetric and reflexive relation).*
- *$\lambda : L \to \Sigma'$ is a labeling function.*
- *$\Sigma' \subseteq \Sigma$ is a set of* restricted events*, giving the scope of an LPO.*

A linearization of a finite LPO is a word of $w_1 \ldots w_n \in \Sigma^*$ such that its canonical LPO $\langle [n], \leq, \{(i, w_i) | i \in [n]\} \rangle$, where $[n]$ is a shortcut for the set $\{1, \ldots, n\}$, is isomorphic to some linear (total) order $\langle L, \leq', \lambda \rangle$ with $\leq \subseteq \leq'$. An ideal is a $\leq$-closed subset of locations, i.e. $\forall l \in I : \forall l' : l' \leq l : l' \in I$. We will abusively call "ideal" the projection of an LPO on a given ideal and allow ourselves to talk about the linearizations of an ideal.

An interpretation of an LPO is a finite or infinite word ($\gamma \in \Sigma^\infty$). An interpretation satisfies an LPO if its restriction to Σ' yields a linearization of the LPO. If the interpretation is an infinite word, it must start with a finite word satisfying the LPO.

Definition 2 ($\models \subseteq \Sigma^\infty \times$ LPO) $\gamma \models \langle L, \leq, \lambda, \Sigma' \rangle$ *iff*

- *$\gamma \in \Sigma^*$ and $\gamma|_{\Sigma'}$ linearizes $\langle L, \leq, \lambda \rangle$,*
- *$\gamma \in \Sigma^\omega$ and $\exists w \in \Sigma^*, \gamma' \in \Sigma^\omega : \gamma = w\gamma'$ and $w \models \langle L, \leq, \lambda, \Sigma' \rangle$.*

There are two versions of LSCs, universal LSCs (uLSC), and existential LSCs (eLSC).

Definition 3 (LSC) *The language of Live Sequence Charts (LSC) is made of uLSCs and eLSCs.*

- *A* universal LSC *(uLSC), with restricted events Σ_R, is a couple of Σ_R-LPOs $\Box(P, M)$, where P is called* prechart *and M is named* main chart*. Remark that we require P and M to be defined over the same alphabet, that we let be Σ_R.*
- *A* existential LSC *(eLSC), with restricted events Σ_R, is a Σ_R-LPO $\Diamond(M)$.*

An interpretation of an LSC is an infinite sequence of events, $\gamma \in \Sigma^\omega$. An interpretation is a model of $\Box(M, P)$ if, whenever P is satisfied in γ, M is also satisfies immediately after.

Definition 4 ($\models \subseteq \Sigma^\omega \times$ LSC) *Let $\gamma \in \Sigma^\omega$.*

- $\gamma \models \Box(P, M)$ *iff*

$$\forall u, v \in \Sigma^*, \gamma' \in \Sigma^\omega : (\gamma = uv\gamma' \text{ and } v \models P) \implies \gamma' \models M.$$

- $\gamma \models \Diamond(M)$ *iff*

$$\exists u \in \Sigma^*, \gamma' \in \Sigma^\omega : \gamma = u\gamma' \text{ and } \gamma' \models M.$$

We lift the notion of model to sets of runs (languages):

Definition 5 ($\models \subseteq 2^{\Sigma^\omega} \times$ LSC) *Let $L \subseteq \Sigma^\omega$.*

- $L \models \Box(P, M)$ *iff for every $\gamma \in L$, $\gamma \models \Box(P, M)$.*
- $L \models \Diamond(M)$ *iff there is some $\gamma \in L$ such that $\gamma \models \Diamond(M)$.*

Since eLSCs are just examples of behavior, they are not as interesting as uLSC for actually specifying protocols. Hence, we will consider that LSC specifications are only made of uLSCs.

Definition 6 (LSC specification) *An LSC specification $\mathcal{S}$ is a finite set of uLSCs. Its model relation is defined as the conjunction of the model relation of its members: $\gamma \models \mathcal{S}$ iff for every $U \in \mathcal{S}$, $\gamma \models U$.*

The size of an LSC is its number of locations. The size of a specification is the sum of the size of its members. A language L is defined by an LSC S if $L \models S$.

A classical question, with respect to logics, is their relation to classes of languages, usually via automata [32,33]. Comparing LSC-definable languages with languages definable in other formalisms determines the expressiveness of LSC. Here, we recall that LSC-definable languages form a strict sub-class of ω-regular languages and a very restricted sub-class indeed. Live Sequence Charts are strictly less expressive than Deterministic Büchi Automata (DBA) [34] and ACTL$^{\text{det}}$, the common fragment of LTL and ACTL [35], as we showed in [36]. In section 4, we will prove that LSCs are exponentially more succinct than DBA and ACTL$^{\text{det}}$. It is possible to translate LSCs to LTL with only a polynomial blow-up. This improves on previous translations that involved an exponential blow-up [8,7]. Another polynomial translation had already been proposed by Kugler *et al.* [37]. Yet, their translation applies only to LSCs in which no event appears twice.

Proposition 7 (From LSCs to LTL) *Any LSC specification $\mathcal{S}$ can be translated to an LTL formula $\phi_{\mathcal{S}}$ with $O(|\mathcal{S}|^5)$ distinct sub-formulae such that*

$$\forall \gamma \in \Sigma^\omega : \gamma \models \phi_{\mathcal{S}} \iff \gamma \models \mathcal{S}.$$

PROOF. We just show how to translate a single uLSC L to an equivalent

LTL formula ϕ_L of size $O(|L|^5)$. The overall formula is just the conjunction $\bigwedge_{L \in \mathcal{S}} \phi_L$. Let $L = \Box(P, M)$.

We define the *index* of a location l in an LPO as l: $idx(l) = |\{l' | \lambda(l) = \lambda(l') \wedge l' \leq l\}|$. A deterministic LPO (DLPO) is an LPO in which locations with similar labels are ordered. Even though DLPO are strictly less expressive than LPO, every (graphical) uLSC can be turned to a model-equivalent DLPO. In a deterministic LPO, by definition, two locations with identical labels have different indexes. Thus, replacing in a DLPO every location l with $(idx(l), \lambda(l))$ results in an isomorphic DLPO. Finally, remark that all linearizations of an LPO have the same length (i.e. exactly the number of locations).

The LTL formula that we build from a uLSC $\Box(M, P)$ is of the form

$$\Box(nprech \vee mainch),$$

where

(1) *nprech* is a formula that asserts that the prechart will not be matched by the subword starting at the current position. It is of the form

$$\bigvee_{l \in P} notoccurs(l) \vee notorder(l),$$

where $notoccurs(l)$ asserts that there will not be $idx(l)$ occurrences of $\lambda(l)$ before having seen $|P|$ occurrences of restricted (Σ_R) events, and *notorder* is a disjunct over all direct predecessors of l. For every direct predecessor l', it says that the number of occurrences of $\lambda(l')$ is smaller than $idx(l')$ when the $idx(l)$-th occurrence of $\lambda(l)$ is encountered. Again, we verify this property within $|P|$ steps. This formula is of size $O(|P|^4)$, because we need 3 counters, ranging over $|P|$, and the outermost disjunction is over all prechart locations.

(2) *mainch* is a formula asserting that, after $|P|$ occurrences of restricted events (i.e. exactly the prechart), for every l and l', where l' is a predecessor of l, l occurs after l' has occurred, yet within $|M|$ steps. Determining the position of l and l' relies on counting $idx(l)$ and $idx(l')$ occurrences of $\lambda(l)$ and $\lambda(l')$, respectively. Again, this formula is of size $O(|C_2|^5)$.

Using this translation, we can rely on the fact that validity, model checking and satisfiability for LTL are all in PSPACE [38], to prove membership of some LSCs-related problems to PSPACE. Those results do not depend on the size of the LTL formula parse tree, but only on the number of its distinct sub-formulae [39]. $\Box$

Every LSC specification is equivalent to the conjunction of liveness and safety properties, *one for every event* in Σ [9]. A scenario S, with restricted events

Σ_R, *forbids* $e \in \Sigma$ after a finite run $w \in \Sigma^*$ iff some suffix of $w|_{\Sigma_R}$, say w', linearizes an ideal I of the LSC, which includes P, but $w' \cdot e$ does not linearize any ideal in S. S *requires* $e \in \Sigma$ iff some suffix w' of $w|_{\Sigma_R}$ linearizes an ideal $I \supseteq P$ of S and $w' \cdot e$ *is* a linearization of some ideal in S.

Definition 8 (forbids, requires) *Let* $\Box(P, M)$ *be a uLSC with restricted events* Σ_R *and* $w \in \Sigma^*$.

- w *forbids* e *iff* $\exists u, v, t \in \Sigma^*$: *such that all the following conditions hold*
  - $uvt = w$,
  - $v \models P$,
  - $\exists I$: *ideal of* $M : I \subset M \wedge t \models I$,
  - $\forall I'$: *ideal of* $M : we \not\models I'$.
- w *requires* e *iff* $\exists u, v, t \in \Sigma^*$: *such that all the following conditions hold*
  - $uvt = w$,
  - $v \models P$,
  - $\exists I$: *ideal of* $M : I \subset M \wedge t \models I$,
  - $\exists I'$: *ideal of* $M : we \not\models I'$.

An infinite run $\gamma \in \Sigma^\omega$ is e-safe iff for every prefix w of this run, if e is forbidden by some scenario after w, we is not a prefix of γ. It is e-live iff for every prefix w of γ, if some scenario requires e after w, then e eventually occurs after w.

The following theorem has been shown in [9] and will be the basis for equipping uLSC with a game-based semantics, hence making it applicable to the specification of agent systems.

Theorem 9 (uLSC = Σ_R-live $\wedge \Sigma_R$-safe) *For every* $\gamma \in \Sigma^\omega$,

$$\gamma \models \Box(P, M) \iff \forall e \in \Sigma : \gamma \text{ is } e\text{-safe and } e\text{-live, wrt } \Box(P, M).$$

2.2 Strategies

Agents are partitioned into two teams: the environment and the system. Formally, $Ag = Sys \,\dot\cup\, Env$. System-controlled events are $\Sigma_{Sys} = Sys \times \mathcal{M} \times Ag$. Engineers are not asked to construct programs for agents in *Env*, only agents from *Sys* have to be implemented. *Sys* implementation will be deployed among *Env* agents that provide thus the model-time context of the specification.

Agents act according to *plans*, or *strategies* [13,40]. Remember that we abstract away from agent's actions and focus on coordination instead. Thus, our abstract view of agent a is a *strategy* $f : \Sigma^* \to 2^{\Sigma_a^s}$. A strategy tells the agent that actions $f(w)$ are advisable to make after some history w. Although this view is very appealing from a mathematical point of view, we will have to

focus on strategies which are representable within computers. We introduce the notion of input/output automata for this purpose.

We will use Input/Output automata to describe the design-time model of agents [41]. An input-output automaton for agent $a \in Ag$ is a finite automaton the alphabet of which is Σ_a. A distinction is made between input events (Σ_a^r) and output events (Σ_a^s) Syntactically, an I/O automaton for agent a must be *input-enabled*: in every state q, agent a should have one transition labeled for every input event. In other words, a may never block incoming messages.

A run of an I/O automaton is an infinite path in the automaton, following the transition relation and starting from the designated initial state. A fair run is a run in which infinitely many transitions labeled with Σ_a^s events are taken. The word generated by a run is the infinite sequence of events encountered along the transitions of the run. The language of an I/O automaton $\mathcal{A}$, denoted $\mathscr{L}(\mathcal{A})$, is the set of words generated by $\mathcal{A}$'s fair runs. The composition of two I/O automata ($\mathcal{A}_1 \times \mathcal{A}_2$) is defined as the synchronous product of $\mathcal{A}_1$ and $\mathcal{A}_2$, see [41] for details.

A finite state I/O automaton represents a finite-memory strategy for agent a. Formally, a (non-deterministic) strategy for agent a is a function $f : \Sigma^* \to 2^{(\Sigma_a^s)}$. It is of finite memory if there is an equivalence relation $\simeq$ on Σ^* such that (1) $\simeq$ is of finite index and (2) $\forall w \simeq w' : f(w) = f(w')$. The size of the memory is the index of the smallest such equivalence relation. Clearly, every finite memory strategy can be translated to an I/O automaton. Conversely, every I/O automaton can be turned into a strategy. The *outcome* of a strategy f is the set of all runs in which Σ_a^s events appear only according to the strategy:

$$Out(f) = \{u_0 e_0 u_1 e_1 \dots | \forall i \geq 0 : u_i \in (\Sigma \setminus \Sigma_a^s)^* \text{ and } e_i \in f(u_0 e_0 \dots u_i)\}.$$

Agents can be organized in societies. A society is a set of agents $A \subseteq Ag$. Its triggered events and sensed events are the union of all triggered/sensed events of its composing agents: $\Sigma_A^s = \bigcup_{a \in A} \Sigma_a^s$ and $\Sigma_A^r = \bigcup_{a \in A} \Sigma_a^r$. The strategy of A is also the union of its agent's strategies: $f_A(w) = \bigcup_{a \in A} f_a(w)$.

We are in position to define when a society of agents is behaving correctly, wrt some given LSC specification. Intuitively, agents within A are only required to respect the specification if agents outside A also do so. For instance, in Fig. 2, if "agenda" is not a system agent, then, other agents are only required to proceed to a vote if "agenda" actually sends a notification. The chairman will thus call for vote *assuming* that other agents are behaving correctly. This is thus very close to the well-known assume/guarantee principle in Computer Science. Thus, agents are only responsible for the correct occurrence of their *own* events.

Definition 10 (Correct Implementation) *A strategy f_{Sys} associated to a society of agents Sys is a* correct implementation *of an LSC specification iff*

$$\forall \gamma \in Out(f_{Sys}) : \begin{cases} \gamma \textit{ is } \Sigma_{Env}\textit{-live} \implies \gamma \textit{ is } \Sigma_{Sys}\textit{-live} \\ \gamma \textit{ is } \Sigma_{Env}\textit{-safe} \implies \gamma \textit{ is } \Sigma_{Sys}\textit{-safe} \end{cases}$$

3 Agent Verification

3.1 Model Checking

In this section, we will investigate the problem of agent verification. Informally, this problem is to check that an implementation of a society is correct. We will consider several consecutive problems. The most general case considers that the society *Sys* consists of at least one agent, and that there might be agents out of *Sys* interacting with them. We will investigate "degenerated" versions, along the following axes:

(1) whether *Sys* consists of a single agent or several agents (viz. centralized vs distributed agent verification);
(2) whether *Env* is empty or not (viz. closed vs open agent verification).

We will start with the simplest problem and progressively consider more difficult ones.

Problem 11 (CCMC) CCMC *(Closed Centralized Model Checking) is the following problem: "Given a strategy f_{Ag}, represented as an I/O Automaton $\mathcal{A}$, and an LSC specification $\mathcal{S}$, decide whether $Out(f_{Ag}) \models \mathcal{S}$."*

Theorem 12 CCMC *is complete for co-NP.*

The hardness proof reduces CCMC to the complement of "Traveling Salesman Problem", which is known to be coNP-complete.

Problem 13 (coTSP) *The* Complement Traveling Salesman Problem *(*coTSP) is to decide whether, for some given constant k, in a given complete graph G, with weights on edges d_{ij}, all tours have a total weight $\geq k$. The weights are all polynomial in $|G|$.

Even with the additional assumption that weights are polynomial in $|G|$, this problem is co-NP complete. Indeed, by inspecting the hardness proof in [42], it actually suffices to consider weights bounded by 2 to obtain co-NP-hardness.

PROOF (Membership) A counter-example is a path in which (i) the prechart is matched and (ii) the main chart never finishes or a safety condition is not met. Such a violation must occur in at most n steps, where n is the number of locations in the Live Sequence Chart. The nondeterministic algorithm guesses the following elements: the LSC L to violate, a state q in $\mathcal{A}$ and a simple path in the synchronous product $\mathcal{A} \times \mathcal{A}_{\neg L}$, with $\mathcal{A}_{\neg L}$ is the linear nondeterministic Büchi automaton recognizing all counter-examples of L. Remark that the simple path is at most of length $n \times |\mathcal{A}|$. $\square$

PROOF (Hardness) There is a polynomial reduction of COMPLEMENT TSP (see [42]) to CCMC. Here, we consider a special case of CCMC, in which all events are system-controlled. A graph G, with a distance d_{ij} is turned into an automaton having states of the form (vertex, counter). The counter sums the weight of the current path, up to the current state. Of course, this counter is bounded by the longest possible path in G. It is thus polynomial in $|G|$, too. The alphabet is the set of vertexes from G. From a state (v, n), there is a transition $(v, n) \xrightarrow{v'} (v', n + d_{qq'})$, iff the edge between q and q' in G has weight $d_{qq'}$. Thus, a path $(v_0, i_0) \dots (v_j, i_j)$ in the automaton corresponds to a path $v_0 \dots v_j$ in G. Furthermore, the total weight of $v_0 \dots v_j$ is i_j.

In any state, there is also a transition to the "down-counting" states: $(v, n) \xrightarrow{\$} (\$, n)$. From these states, the automaton counts down, decreasing the counter by one unit at a time, until its counter equals 0: for $n > 0$, $(\$, n) \xrightarrow{\text{tick}} (\$, n-1)$. When zero is reached, the automaton reads an infinite sequence of "end" events: $(\$, 0) \xrightarrow{\text{end}} (\$, 0)$. Finally, we add an initial state q_0, with a transition $q_0 \xrightarrow{\text{init}} (q, 0)$, for all q. This automaton has $2 + D \cdot (|G| + 1)$ states, where D is the maximal distance. It is thus polynomial in the size of the original graph.

The fact that all tours have length $\geq k$ is encoded in an LSC as follows: the prechart contains $\{q_1, \dots, q_n, \$\}$, where q_i's are unordered, whereas \$ is greater than all q_i.

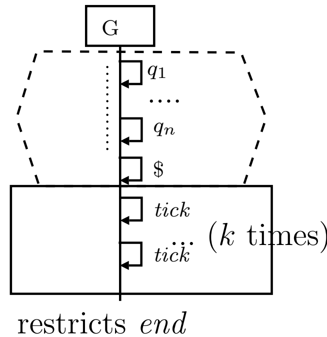


Fig. 8. All tours have length $\geq k$

The prechart is matched when all vertexes have occurred exactly once and, then, the automaton has announced that it will start down-counting. Then, the main chart checks that tick occurs k times, without any end event in between. It is easy to see that there is a tour of total weight $< k$ iff the automaton violates the LSC, i.e. the prechart is matched (we found a tour), but the main chart is violated afterwards. Violating the main chart means that, before k ticks, the "end" event occurs. Hence, the total weight of the tour is smaller than k. $\square$

A first extension to this problem is to consider that some agents belong to the environment, while others are system agents. Then, we are presented with an implementation of system agents only and the question becomes: "whenever environment agents do behave correctly, does this implementation behave appropriately?".

Problem 14 (OCMC) OCMC *(Open Centralized Model Checking) is the following problem: "Given a partition of Ag into Sys and Env, a strategy f_{Sys}, represented by $\mathcal{A}$, and an LSC specification $\mathcal{S}$, decide whether f_{Sys} is a correct implementation of $\mathcal{S}$ (see def. 10)."*

Theorem 15 OCMC *is complete for PSPACE.*

The proof of this theorem is similar to the proof provided in Sec. 3.2. The computations of a DPSPACE Turing Machine can be encoded in an LSC specification, in polynomial-time and logarithmic space. The automaton generates only traces starting with an initialization event and, eventually, emitting a halting event.

The second restriction imposes that we consider monolithic systems only, made of a single component. As it was clear from the introduction, we are mostly interested in distribution systems. The design-time specification of such systems will typically be presented as a "network" of automata, one for each agent. Every automaton prescribes how its owner shall behave, see Sec. 2.2.

Problem 16 (CDMC) CDMC *(Closed Distributed Model Checking) is stated as follows: "Given an LSC L, a list of strategies $(f_a)_{a\in Ag}$, represented by $(\mathcal{A}_a)_{a\in Ag}$, decide whether $Out(f_{Ag}) \models L$.*

Unfortunately, as usual in verification [43], distribution makes model checking more complex. Now, the problem becomes PSPACE-complete instead of coNP-complete. Remark that we present, in CDMC, a degenerated problem, for only *one* scenario is used in the specification. Considering an actual specification is not harder. Actually, there is an immediate nondeterministic PSPACE algorithm deciding the complement of the problem: pick nondeterministically one scenario in the specification and check that the implementa-

tion violates it. This problem is exactly the complement of CDMC, which is thus in coPSPACE=PSPACE, by Savitch's theorem [44].

Theorem 17 CDMC *is PSPACE-complete.*

PROOF (Membership) Let m be the size of $\mathcal{A}_i$'s and the LSC be of size n. By Savitch's theorem, it suffices to build a nondeterministic PSPACE Turing machine deciding the complement of the distributed model checking problem. This algorithm guesses an initial state and a path in the product of the automata. As this path needs to be ultimately periodic, it also guesses the following elements: the index in the path at which the loop is entered and the length of the path, as in [38]. We then check that the transition relation of the LSC is correctly followed , thus only two configurations need to be saved, plus the configuration at the entry of the loop. Within the loop, either no environment event occurs, but no such event is required, or some event occurs infinitely often. □

PROOF (Hardness) Consider an arbitrary DPSPACE Turing machine. Assume that its set of control locations is Γ and its symbols are Σ. One can without loss of generality, assume that the machine uses only its input space. Otherwise, the input can be padded with n^k blank spaces, see In-Place Acceptance in [42]. For every cell tape, we build an automaton, say $\mathcal{A}_i$. The alphabet of the system is $\{\text{init}, \text{halt}\} \cup (\Gamma \times \{1, \ldots, n\})$. An event (γ, i) means that the tape head moves to cell i and the control location becomes γ. $\mathcal{A}_i$ has two types of control locations, to record the fact that the tape head is on its cell or not. The former is of the form $(a, \gamma) \in \Sigma \times \Gamma$ and the latter of the form $a \in \Sigma$. Assume that we want to encode a transition $(\gamma, a, r, a', \gamma')$, i.e. when the TM control location is γ and it reads a from the cell on which the tape head resides, the TM writes a', moves the tape head to the right and the control location becomes γ', of the Turing machine. Let the tape head be on cell i. Then, $\mathcal{A}_i$ will contain a transition $((a, \gamma), (\gamma', i + 1), a')$, while $\mathcal{A}_{i+1}$ has a transition $(b, (\gamma', i + 1), (b, \gamma'))$. All automata synchronize on a first common event "init". The "init" event is caught by the prechart. The main chart then asserts that "halt" will eventually occur. □

Combining distribution and openness does not increase the problem complexity; it is still PSPACE-complete.

Theorem 18 ODMC *is PSPACE-complete.*

One could believe that this high complexity is due to the presence of automata in the problems. Actually, reachability in networks of automata is already a difficult problem [42], as hinted to by Theorem 17. The next section presents

simple analysis problems, on LSCs only, that are also difficult. This is amazing, as one might think that these problems can be solved by easy computations on the diagrammatic form of LSCs.

3.2 Reachability and Refinement Checking

The first problem we consider is whether an LSC specification is compatible with an existential LSC.

Problem 19 (REACHABILITY) *Given an eLSC L and an LSC specification $\mathcal{S}$, decide whether*

$$\mathscr{L}(\mathcal{S}) \models L.$$

REACHABILITY checks that a certain specification, together with assumptions over the domain still makes it possible to achieve a certain behavior. In software engineering terms, REACHABILITY is used when one wants to check that the future system specification does not disallow a certain use case. We have just seen that this problem was PSPACE complete. Using the same idea of reduction, one can show that specification refinement is also PSPACE complete. Verifying specification refinement is a natural problem, in the framework of a progressive software development approach. Given a certain abstract specification $\mathcal{S}$, a more precise specification $\mathcal{S}'$ is designed and we want to verify that every behavior induced by $\mathcal{S}'$ is a legal behavior of $\mathcal{S}$. Logically, this boils down to verifying the validity of $\mathcal{S}' \rightarrow \mathcal{S}$.

Problem 20 (LSC-IMPL) *The problem of implication of LSC specifications (*LSC-IMPL*) is given two LSC specifications $\mathcal{S}$ and $\mathcal{S}'$, to decide whether*

$$\forall \gamma \in \Sigma^\omega : \gamma \models \mathcal{S} \implies \gamma \models \mathcal{S}'.$$

Satisfiability of LSC specifications is polynomial-time reducible to reachability. One can add a scenario obliging the machine to perform an infinity of computations: every time it reaches the halting location, it is launched again, from the init location. Hence, only runs in which the machine can "halt" from the initial location will be models of the specification.

Problem 21 (LSC-SAT) *The problem of LSC satisfiability (*LSC-SAT*) is to decide, given an LSC specification $\mathcal{S}$, whether*

$$\exists \gamma \in \Sigma^\omega : \gamma \models \mathcal{S}.$$

Hence, the two problems also considered in this section are as difficult as reachability. This is not surprising as reachability is an important primitive of

most verification algorithms.

Theorem 22 REACHABILITY *is complete for PSPACE.*

Corollary 23 LSC-SAT *and* LSC-IMPL *are PSPACE-complete.*

PROOF (Membership) LSCs can be transformed in polynomial time, using logarithmic space, into equivalent LTL formulae of the same size. Let their conjunction be Φ_u. The existential LSC can be turned into a "never claim" LTL formula, claiming that the existential LSC is never matched, that we denote ϕ_e. Then, we ask whether the formula $\Phi_u \to \phi_e$ is valid. This is true iff the existential LSC is unreachable, i.e. this solves exactly the complement of our problem. The solution via LTL is in PSPACE, see [38]. This class being closed under complement, we have that REACHABILITY is in PSPACE, too. □

PROOF (Hardness) We encode the execution of a DPSPACE Turing Machine on the blank input within an LSC specification. Assume that the control locations of the TM are taken from a a finite set Γ. Furthermore, suppose that the TM has been modified in such a way that, when it moves the tape head beyond the input, it loops forever in some non-halting state. We let the alphabet of the tape cell be the binary alphabet $\{0, 1\}$. Finally, we suppose that, among Γ, the halting location is γ_h, which is never left once it is reached. Since it is a DPSPACE TM, it uses at most n cells of memory. The run of the TM will be encoded as an infinite word over the alphabet:

$$(\Gamma \cup \{in, \$\} \cup \{0, 1\}) \times \{0, \dots, n\}.$$

The LSC specification contains only one agent; we will thus omit it in the rest of the proof. A correct encoding will have the following form $init \cdot exec$, where

$$init = (in, 0)(0, 0)(in, 1)(0, 1) \dots (in, j)(0, j) \dots (in, n)(0, n)(\gamma_0, 0) \qquad (1)$$

The "init" sequence ensures that, at the beginning of the run, the tape cell contains n blank cells and the initial location is γ_0, with the tape head on cell 0. An event (in, j) requires the agent to perform $(0, j)$, i.e. to immediately initialize the j-th tape cell to 0.

We express this "initialization sequence" using the LSCs in Fig.9, which restrict *all* events.

Consider an arbitrary configuration of the TM: $C = (T, \gamma, i)$, where T is the tape content, γ is the control location and i is the tape head position. We say that it is encoded by a word w if

(1) $\exists v : w = v(\gamma, i)$

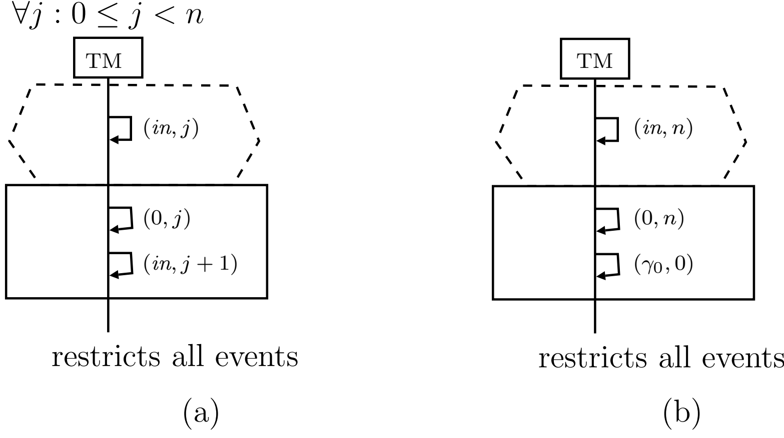


Fig. 9. Initialization Sequence of DPSPACE TM

(2) $\forall j : 1 \leq j \leq n : T[j] = a \implies \exists u, v : w = u(a, j)v$ and neither $(0, j)$ nor $(1, j)$ appears in v.

Notice that, when w fulfills these conditions, C can be unambiguously retrieved from w.

It is easy to check that *init* encodes the initial configuration $C_0 = (T_0, \gamma_0, 0)$, where $T_0[j] = 0$, for all j. We need to express the successor relation between two configurations $C \vdash C'$.

Suppose, wlog, that $C = (T, \gamma, i)$, $T[i] = 0$ and $C' = (T', \gamma', i+1)$, where T' is like T, except that 1 has been written at the i-th position. Assume that C is encoded by some word w. By definition of configuration encoding, $w = v \cdot (\gamma, i)$, and the last occurrence of either $\{(0, i), (1, i)\}$ is $(0, i)$ in w. The transition will be encoded as the following continuation:

$$w' = v \underbrace{(\gamma, i)(0, i)(\$, i)(1, i)(\gamma, i+1)}_{u}.$$

One can check that w' is indeed an encoding of C', by noting that

(1) it ends with $(\gamma, i+1)$;
(2) in u, no event of the form $(0, j)$ or $(1, j)$ $(j \neq i)$ has been added. Hence, the tape content of the configuration encoded by w does not differ from that of C on these cells.

The proof is almost over, we simply need to describe all sequences of the form above with a conjunction of LSCs. This is achieved with the scenarios of Fig. 9 to 12. The first one retrieves the last occurrence of an event of the form $(0, i)$ or $(1, i)$. It is copied immediately after (γ, i). This retrieval is presented in Fig.10. One should take care of a detail, here: we want to be sure that after (γ, i), only *one* occurrence of $(0, i)$ will be repeated. This is achieved by using *no-scenarios*, the prechart asserts that matching a sequence of the

form $(a,i)(a',i)(\$,i)$, where $a, a' \in \{0,1\}$, should cause a contradiction in the specification. Therefore, such a "bad" encoding is forbidden.

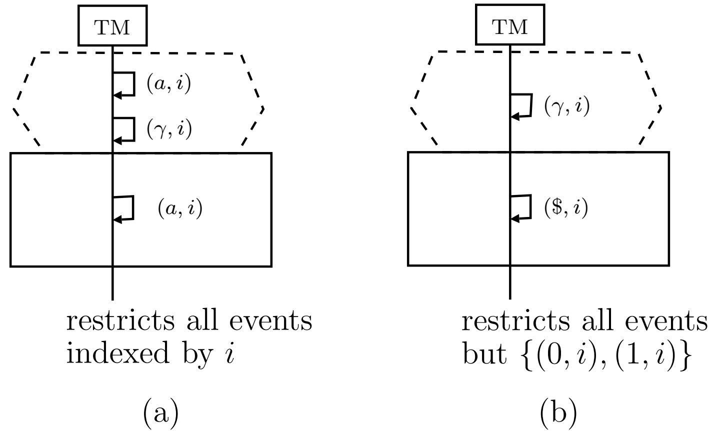


Fig. 10. Retrieving tape cell content

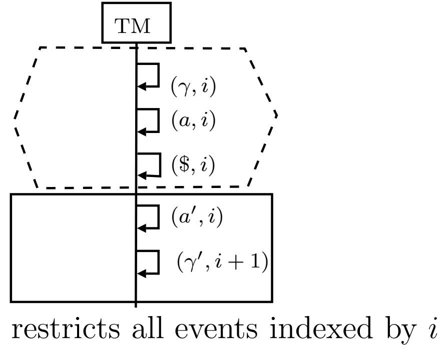


Fig. 11. Transition $(\gamma, a, a', r, \gamma')$

A third scenario encodes the rest of the transition, i.e. writing to the i-th cell and moving the tape head to the right. This scenario is shown in Fig.11.

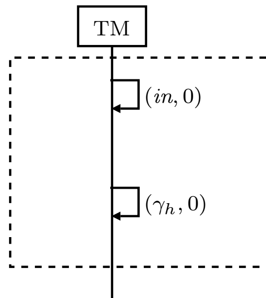


Fig. 12. Existential scenario: TM initializes and eventually halts.

To conclude, we use the existential LSC to encode the property that, after having been initialized, the TM eventually halts. This scenario, in Fig.12, ignores all events, but the two in it. □

In 2001, Harel and Marelly introduced an algorithm and an approach to the validation of LSC-based specifications, called *play-out* [45]. The specification

is immediately executed, without generating any code from it, but using an animation engine instead. This animation engine uses a super-step approach: when the environment inputs some new event, by performing some action on the graphical user interface, the engine performs all system-controlled events that become required, until it reaches some stable status, in which no event is required anymore. The theorems provided in this section can be adapted to show that (1) computing whether a finite super-step exists is PSPACE-complete and (2) the animation process (even if there is a single possible sequence of events) is not space-efficient, as it can simulate a DPSPACE Turing Machine.

4 Agent Design

In this section, we turn to the most complex class of problems considered in this paper. We want to determine whether agents can indeed be implemented in order to satisfy the protocol. Ideally, the proof of implementability should be constructive: some strategy, for every agent, must be built. Would this implementation be compact and readable, the burden of designing the system would be taken away from engineers. This achieves Harel's "achievable dream" [46].

As in the previous section, we will consider two versions of this problem. The first version requires us to build a strategy for *Sys*, say f_{Sys}, which is represented as a single automaton $\mathcal{A}$. The second version, that we call "distributed", obliges us to find a "distribution" of f_{Sys} into $(f_a)_{a \in Sys}$. This problem turns out to be undecidable.

Remark that we will not be considering the problem of designing *closed* agent design. This is because this problem is rather trivial. It suffices to test whether $\mathscr{L}(\mathcal{S})$ is nonempty, which is formally equivalent to LSC-SAT.

We are more interested in the design of open agent systems. They are going to be deployed in adversarial environments. Under these conditions, the problem of implementability is not equivalent to satisfiability [47]. The question is more accurately posed as "is there an implementation of system agents such that, no matter how environment agents behave, the specification will be respected?".

Problem 24 (COAD) COAD *(Centralized Agent Design) is the problem of deciding, given an LSC specification $\mathcal{S}$ and a set of system agents $Sys \subseteq Ag$, whether there is a strategy f_{Sys} such that f is a correct implementation of $\{L_1, \ldots, L_m\}$."*

In [9], we have presented an exponential time algorithm solving COAD. It

constructs a two-player parity game graph, with three colors, in which player 0 has a winning strategy iff the specification is realizable. The game graph is exponentially larger than the LSC specification. Solving a parity game with three colors can be done in quadratic time [48].

This problem is EXPTIME-complete. This proves our claim that, because LSCs are less expressive than LTL, some problems are easier on LSCs than on LTL. Actually, centralized realizability is 2EXPTIME-complete for LTL [49].

Theorem 25 COAD *is complete for EXPTIME.*

PROOF (Membership) The algorithm presented in [9] builds a two-player parity game graph, with 3 colors, from an LSC specification. The game graph has size $2^{O(n \log n)}$, where n is the size of the specification. The first player (protagonist) has a winning strategy on this game graph if, and only if, the specification is consistent. This generalizes the approach presented in [7]. □

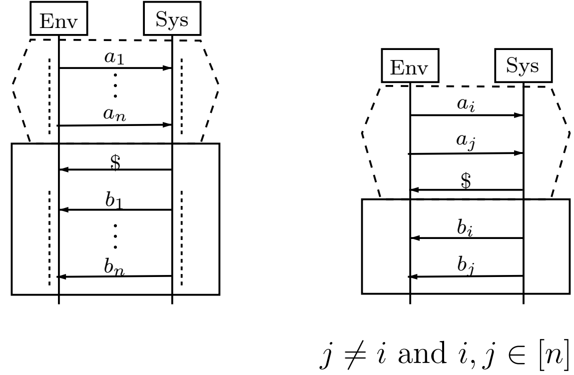
PROOF (Hardness) We encode an alternating PSPACE Turing machine into an LSC, as we did before (see Th. 22). The result will follow from the fact that APSPACE=EXPTIME [50]. The only difference is that we need to distinguish between universal and existential moves of the machine. Since alternation is built in the realizability problem, we can use the two statuses of the player to model the alternation of the Turing machine. In order to do so, we duplicate all events, and assign them to player 0 and player 1. A transition is now of the form $(\gamma, i, A)(a, i, A)\$_i(a, i, A)(\gamma', j, A')$, where $A, A' \in \{\forall, \exists\}$ indicates the status of the current state (universal or existential).

Since there are several possible moves at configurations (by definition of alternation), we need to encode these possible continuations. All bad continuations are encoded in no-scenarios, which imply contradictory requirements on the player $(\forall, \exists)$ who is about to play. Thus, if this player decides to pick such a bad continuation, the outcome will certainly not respect the LSC specification. This is equivalent to complete "a priori" the TM transition relation, without altering its language.

We add anti-scenarios, to ensure that player i loses as soon as he performs a move when it is not expected to do so. Surprisingly, we assign existential moves to player 1 and universal moves to player 0. A scenario is added, ensuring that player 0 loses as soon as a halting configuration is met. The specification is not realizable iff the machine has an accepting computation. Actually, player 1 can pick existential moves such that the computation tree halts on all its paths (otherwise, player 0 would have a winning strategy to escape). □

The algorithm presented in [9] is computationally expensive, yet optimal. However, it suffers from another problem: it yields design models, as automata, that are exponentially larger than the specification. This is a hindrance for readability. Nevertheless, we show below that strategies realizing LSC specifications need memories that large. Therefore, our algorithm is optimal, in the sense that every algorithm solving this problem will necessarily build exponentially large implementations.

We exhibit in Fig 13 a family of LSC specifications $(\phi_n)_{n>0}$the size of which grows quadratically in n but any strategy for *Sys* realizing ϕ_n



$j \neq i$ and $i, j \in [n]$.

Fig. 13. LSC specification ϕ_n

In this game, *Env* controls $\{a_1, \ldots, a_n\} = \Sigma^s_{Env} = \Sigma^r_{Sys}$ and *Sys* controls $\{\$, b_1, \ldots, b_n\} = \Sigma^s_{Sys} = \Sigma^r_{Env}$. *Env* first presents *Sys* with a sequence of n symbols. Remark that *Env chooses* the order in which those events occur. When the whole sequence has been presented, *Sys* must reply with the same sequence. Hence, *Sys*'s strategy must have at least enough memory to remember the order in which the n events have been presented. The LSC specification encoding this is presented in Fig. 13. Along "Sys" and "Env" on the left-hand side scenario, we drew two dashed lines. This defines a *co-region*, which relaxes the ordering on the enclosed events. Therefore, $a_1 \ldots a_n$ can occur in any order, see Section 2.1. In comparison, on the right-hand side, a_j and a_i are ordered. The right-hand side scenario obliges b_j to follow b_i if a_j occurred after a_i.

Theorem 26 (Memory Lower-Bound) *There is a family of LSCs specification, namely $(\phi_n)_{n>0}$ such that any strategy realizing ϕ_n has a memory of size $2^{\Omega(n \log n)}$.*

PROOF. First of all, for every n, $|\phi_n| = 5n^2 + 3n + 1$. Hence, the size of ϕ_n grows only quadratically in n.

Now, consider some strategy $f : \Sigma^* \to \Sigma^s_{Sys}$ winning in this game. If f is a correct implementation, it *must* have enough memory to remember the order

in which $a_1 \dots a_n$ occurred. Otherwise, there would exist two words w and w' of $(\Sigma^s_{Env})^*$ such that $elts(w) = elts(w') = \Sigma^s_{Env}$ but f cannot distinguish between them, i.e. $w \simeq w'$, and thus $f(w) = f(w')$ (see Sec. 2.2). However, $w \cdot f(w) = w \cdot f(w')$ and consequently, f would not be winning, since the order of replies (b's) does not match does the order of queries (a's). Contradiction.

All permutations of $a_1 \dots a_n$ are possible, therefore there must be as many memory states in f as there are permutations of n elements, i.e. $2^{\Omega(n \log n)}$. □

Remark 27 (Succinctness) *Using the same family of LSC specifications and the same proof, one can show that translating LSCs to some DBA involves an exponential blow-up. Actually, it is not even possible to translate LSCs to NBA recognizing either the language of the specification or its complement without this blow-up. It follows from this fact and from the theorems in [35] that turning LSCs to equivalent $ACTL^{det}$ formulae also involves an exponential blow-up. Indeed, for every $ACTL^{det}$ formula, there is a nondeterministic Büchi automaton recognizing their complement, which is linear in their size.*

The problem of centralized realizability is lacking some features, which lessens its applicability

(1) It would be interesting to come up with an implementation which satisfies the specification and guarantees that additional requirements will be met as well. This is especially interesting if the specification is too abstract or too loosely defined to ensure the requirements, but the analyst thinks that it is possible to refine it in a way that would fulfill the requirements. The problem of deciding whether there is such a particular implementation, which we call *constrained* centralized agent design is 2EXPTIME-complete, when we consider LTL as a language for expressing requirements.
(2) It does not take agent interfaces into account, because it assumes that the "perfect information" hypothesis holds. Hence, agents are not obliged to consider only events occurring at their interfaces. It seems necessary to extend the centralized version of the problem to take this into account. This variant is called *distributed* agent design. As for LTL, this problem is undecidable [51].

Problem 28 (LTL-CONS-COAD) *The problem of LTL-Constrained Centralized Open Agent Design (*LTL-CONS-COAD*) is, given an LSC specification $\mathcal{S}$, a set of system agents $Sys \subseteq Ag$ and an LTL formula φ, to decide whether there is a strategy $f_{Sys} : \Sigma^* \to \Sigma^s_{Sys}$, such that*

(1) f_{Sys} is a correct implementation of $\mathcal{S}$;
(2) $Out(f) \models \varphi$.

Theorem 29 LTL-CONS-COAD *is complete for 2EXPTIME*

PROOF. Membership is easy: translate LSCs to LTL (Prop. 7) and check LTL realizability in doubly-exponential time [49]. Hardness comes from the fact that this problem generalizes LTL realizability, which is 2EXPTIME complete [49]. □

The problem of distributed agent design is to build a strategy for every agent in a society such that

(1) agents respect their *interfaces*, i.e. agent a senses events from Σ_a^r only.
(2) the society is well-behaving, with respect to an LSC specification.

Surprisingly, this problem is undecidable. Furthermore, the proof uses LSCs without any fancy constructs: no loops, no alternatives, no conditions, ...

Problem 30 (DOAD) *The* DOAD *(Distributed Open Agents Design) problem is defined as: "Given an LSC specification $\mathcal{S}$ and a society of agents Sys, decide whether there is a list of strategies $(f_a)_{a \in Sys}$ one for every system agent, such that*

(1) $f_a : \Sigma^* \to (\Sigma_a^s)$;
(2) $\forall w, w' \in \Sigma^* : w|_{\Sigma_a} = w'|_{\Sigma_a} \implies f(w) = f(w')$, *i.e. if w and w' are the same, from a's point of view, then a shall behave the same way after w or w';*
(3) f_{Sys} *is a correct implementation of* $\mathcal{S}$.

Theorem 31 DOAD *is undecidable.*

PROOF. We reduce Post's Correspondence Problem (PCP) to the problem of deciding whether the specification is not implementable, following [52].

We first recall the definition of PCP. A PCP instance is a list of pairs of words (w_1, u_1) , ..., (w_n, u_n), such that, for all i, $w_i \neq u_i$ and $w_i, u_i \in \Theta^*$ (for some finite alphabet Θ). A *solution to a PCP instance* is a finite sequence of indexes $i_1 \dots i_m$ ($m \geq 1$ and $1 \leq i_j \leq n$, for all j) such that $w_{i_1} w_{i_2} \dots w_{i_m} = u_{i_1} u_{i_2} \dots u_{i_m}$. The problem of telling whether any PCP instance admits a solution or not is undecidable.

Let us fix an arbitrary PCP instance. We show how to reduce the problem of determining whether this PCP instance admits a solution to DOAD. The alphabet of our LSC specification is $\Theta \cup \{k_1, \dots, k_n\} \cup \{\$\} \cup \{0, 1\} \cup \{A_0, A_1\}$, plus an arbitrary finite number of events that can be exchanged between system agents, say $\{s_0, \dots, s_q\}$. The system is made of two agents: a_1 and a_2. The first agent may observe $\Theta \cup \{\$\}$, whereas the second can observe $\{k_1, \dots, k_m, \$\}$. All these events, but $\{A_0, A_1\}$ and the additional system events $\{s_0, \dots s_k\}$

are controlled by the environment. A play proceeds as follows. First, the environment picks either 0 or 1. The former means that the environment chooses to read words in the first component of the pairs of words (viz. the w_i's), the latter means that it will read u_i's. Then, the environment must stick to that choice until the end of the play. Namely, the environment chooses a particular word in the list (say, w_i or u_i, depending on the "column" chosen) and indicates the index of this word to the system, by performing k_i. The environment must then enumerate the letters in w_i, which are thus published to agent a_1. The game goes on until the environment performs \$. At this point, the system is required to output A_0 or A_1, depending on what index (0 or 1) the environment had chosen in the first place.

We claim that the PCP instance has a solution iff this specification is not implementable. Assume that PCP has a solution $i_1 \ldots i_m$ but there is a winning strategy for the system. Then, upon $0 i_1 w_1 \ldots i_m w_m \$$, the system answers with 0. Nevertheless, the strategy of the system shall also answer 0 to $1 i_1 u_i \ldots i_m u_m \$$, because the projection of the two words on agent's alphabets are the same. Therefore, there is no winning strategy.

If PCP has no solution, then, the two system agents can get together and compare the submitted run. Agent a_2 sends the sequence of indexes that it has been presented with to a_1 (using some protocol on which they agreed, based on $\{s_0, \ldots, s_p\}$). This agent can then build $w_{i_1} \ldots w_{i_m}$ and compare it with the word that he has received from the environment. Since PCP has no solution, either they are the same and a_1 shall answer 0 or the two words differ and a_1 replies with 1. □

5 Extensions

5.1 Control Flow

The language of LSC that we have used so far was pretty simple. In this section, we present some possible extensions, that make it more expressive but does not cause any changes in the complexity of the problems investigated in this paper. Actually, all membership proofs can be simply adapted to deal with these extensions. Hardness proofs are of course not affected by adding new constructs to the language.

Alternatives: within a single LSC, one can describe several alternatives, as is done with in-line constructs of MSCs or AUML Interaction Diagrams. We need to introduce the concept of LPOs with choice, which is much heavier to manipulate. This extension does not cause any problem, besides

to our translation to LTL, which is not correct anymore, because the length of LPOs is variable. Our translation relies crucially on the fact that all linearizations of an LPO have the same length.

Conditions: it is possible to add conditions (i.e. boolean logic over some predefined set of propositions), to the language. Together with alternatives, we can embed if-then-else tests in the language. Using the concept of cold/hot conditions, one can also describe some "preconditions" and assertions: a hot condition describes a condition that must be true when it is evaluated, whereas a cold condition represents a condition that, if evaluated to false, finishes prematurely and successfully the scenario. Again, all the results of this paper remain true if we consider this extension. If we have only "hot" conditions, the translation to LTL still works.

Hot/Cold Locations: a cold location is a location on which the execution of the chart may stop. This provides us with a way to specify that some linearizations of the LPO may stop before reaching its end. All complexity results are preserved by this extension, except for the translation to LTL, because the length of the LPO is now variable.

Modes of communication: In our model, we assumed that communication was instantaneous. Nevertheless, we can represent other modes of communication, like asynchronous or synchronous communication in our model. Asynchronous communication means that the receiver shall not be ready for the sender to send its message. In the synchronous mode, there is a transmission delay, too, but the sender must wait for the receiver to get the message before proceeding. This represents procedure calls, in programming languages.

Unbounded loop is the only extension for which we could not prove the robustness of our constructions. With the Kleene star and alternatives, we can encode every regular expression as a basic chart. We were not able to show that the double blow up involved in the tableau method could be avoided, and we leave that problem open. Remark that Kleene star makes the language incomparable to LTL.

5.2 Roles

Symbolic LSCs, which have been informally introduced in section 2.1, makes it possible to describe the behavior of unbounded families of agents. We introduce *roles* in our approach. In logical terms, Symbolic LSCs are to LSCs what first-order logic is to propositional logic. We follow as much as possible [27], even though their solution has been tuned for animation, and its formalization might not sound as clean as it could be.

Role is a set of roles. A population is a partial function, with *finite domain*,

$Pop : Ag \nrightarrow 2^{Role}$, mapping every agent to the roles he plays. Let $\mathbb{P}$ denote the set of all populations. We assume here that agents may not change roles during system execution. We also drop the hypothesis that Ag is finite and only require it to be countable.

We also assume that we are given a countable set of first-order variables, *Var*. *Var* and Ag are distinct. An interpretation of $V \subset Var$ is a function $\theta : V \rightarrow Ag$. Message terms are also extended to include first-order variables. We write $\Sigma(V)$, for $V \subset Var$, to denote the set $(Ag \cup V) \times \mathcal{M} \times (Ag \cup V)$. Ground events are events from $\Sigma(\emptyset)$. Applying a V-interpretation to an event in $\Sigma(V)$ yields a ground event, in which all occurrences of $v \in V$ is replaced by $\theta(v)$. Let $\mathbb{I}$ represent the set of all interpretations.

In the same vein, we extend LPOs, and transform them in *Quantified Labeled Partial Order* (QLPO). A QLPO is an LPO over $\Sigma'(V)$, or an expression of the form $\forall x : R : Q$ or $\exists x : R : Q$, where $x \in Var$, $R \in Role$ and Q is a QLPO, in which x is a free variable. We use the usual definition of *free* and *bound* variable. A variable is bound if it occurs within the scope of a quantifier. It is free if it is not bound.

Applying an interpretation of variables to an LPO simply replaces all occurrences of variables by their interpretations in event terms ($\Sigma(V)$). If all free variables of an LPO are interpreted in θ, this yields a ground LPO, as well.

Definition 32 ($\models \subseteq \mathbb{P} \times \mathbb{I} \times \Sigma^\infty \times$ QLPO) *Let $\gamma \in \Sigma^\infty$, Pop is a population and θ is a first-order variable interpretation.*

- *$Pop, \theta, \gamma \models Q$, with $Q \in LPO$ iff $\gamma \models \theta(Q)$.*
- *$Pop, \theta, \gamma \models \forall x : R : Q$ iff, for every $a \in Agents$,*

$$R \in Pop(a) \implies Pop, \theta \cup \{x \mapsto a\}, \gamma \models Q.$$

- *$Pop, \theta, \gamma \models \exists x : R : Q$ iff, there is some $a \in Agents$, such that*

$$R \in Pop(a) \text{ and } Pop, \theta \cup \{x \mapsto a\}, \gamma \models Q.$$

A Symbolic uLSC is a pair $\Box(P, M)$ such that

(1) P is a $\Sigma'(V)$-LPO. Thus, all variables in V are free in P. We do not allow quantifiers in the prechart, as is also done by [27].
(2) M is a $\Sigma''(V')$-LPO, with $\Sigma''(V') \supseteq \Sigma'(V)$, in which the sole free variables are V.

Symbolic LSCs are interpreted against populations and infinite words $\gamma \in \Sigma^\omega$. An interpretation satisfies a Symbolic LSC if, whenever the prechart is matched, the main chart is also matched afterwards. Remark that matching

can be done according to several variable interpretations, and we take all of them into account.

Definition 33 **($\models \subseteq \mathbb{P} \times \mathbb{I} \times \Sigma^\omega \times$ SymLSC)** *$Pop, \gamma \models \Box(P, M)$ iff, for every first-order variable interpretation θ, for every decomposition $uv\gamma'$ of γ,*

$$Pop, \theta, v \models P \quad \Longrightarrow \quad Pop, \theta, \gamma' \models M.$$

A Symbolic LSC specification $\mathcal{S}$ is a finite collection of Symbolic LSCs. As for plain uLSCs, the semantics of a specification is defined through conjunction:

$$Pop, \gamma \models \mathcal{S} \iff \forall S \in \mathcal{S} : Pop, \gamma \models S.$$

Problem 34 **(SymLSC-SAT)** *The satisfiability problem for Symbolic LSCs* SymLSC-SAT *is given a Symbolic LSC specification $\mathcal{S}$ and a finite set Role, to decide whether there is a finite population $Pop : Ag \not\to 2^{Role}$ such that*

$$\exists \gamma \in \Sigma^\omega : Pop, \gamma \models \mathcal{S}$$

Theorem 35 SymLSC-SAT *is undecidable.*

PROOF. We outline how one can reduce the halting problem of a two-counter machine, which is known to be undecidable, to SymLSC-SAT. A two-counter machine (2CM) is a program (i.e. a finite list *prog*), that has two integer counters c_0, c_1 and uses the following statements:

- `init` is an *initialization* statement, that resets c_0 and c_1 to 0. There is only one `init` statement, located at line 0 of *prog*.
- `go to` l_1 `or` l_2, where l_1 and l_2 are line numbers, with $l_1, l_2 > 0$. Executions must jump (nondeterministically) at line l_1 or l_2.
- `halt` is a *halting* statement. There is only one `halt` statement. Its effect is to make the execution back to line zero, i.e. to the `init` statement.
- `inc` i, with $i = 0, 1$. Its effect is to increment counter c_i of one unit and goes on with the statement at the next program line.
- `dec` i decrements c_i and goes on with the statement at the next program line.
- `not` i. The execution goes on if $c_i \neq 0$. Otherwise, the execution stops.
- `zero` i. The execution continues if $c_i = 0$, otherwise, it stops.

The problem of deciding, given *prog*, if there is an execution that will eventually execute `halt` is undecidable. Remark that, if *prog* executes `halt`, then there are two bounds $k_0, k_1 \in \mathbb{N}$ such that $c_i < k_i$ $(i = 0, 1)$, during the whole execution.

By construction, it is easy to see that

(1) determining whether there is an infinite execution that goes infinitely often through `init` is undecidable, too. Actually, the same finite execution, from `init` to `halt` can be iterated again and again.
(2) if there is such an ever-looping execution, it also uses counter bounds k_0 and k_1.

In order to encode counter values with Symbolic LSCs, we use agent roles. In our case, $Role = \{cntr\}$. Every agent playing role *cntr* can assume three "values": -1 (meaning unused), 0 and 1. The value of counter $c_i (i = 0, 1)$ is the number of agents assuming value i. We also use a concrete instance, named "cpu", which is a "central processing unit". It executes sequentially the 2CM statements as prescribed by *prog* and sets the values of *cntr* agents. Agent *cntr* can receive four messages: "get", "unset", "set0" and "set1". The first one queries the value currently stored (-1, 0 or 1, thus). The three last messages set the value.

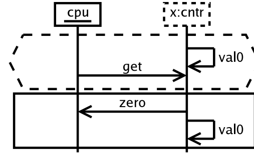


Fig. 14. Getting x values

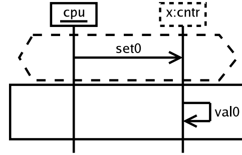


Fig. 15. Setting x value

The LSC of Fig. 16 encodes the semantics of `init`: it sets c_1 and c_0 to 0, by ensuring that there are no *cntr* agents with values 1 or -1. Then, it proceeds to the next statement, which is at line number 1.

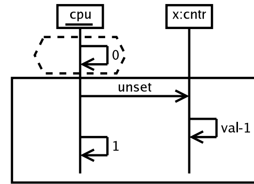


Fig. 16. `init`

The CPU sends to itself the line number of the next statement to execute. If line i is a statement of the form `inc` 1, this line is translated to the LSC of Fig. 17. In this LSC, some agent in *cntr* is picked, the value of which is -1 (i.e. it does not belong to any counter), and sets its value to 1. Since all other agents do not take part in this protocol, their value is unchanged. Remark that the execution proceeds at the next line, i.e. $i + 1$.

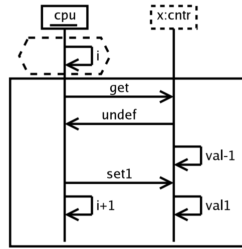


Fig. 17. `inc 1`

The same approach is taken to translate the statement `dec` 1. This is illustrated by Fig. 18.

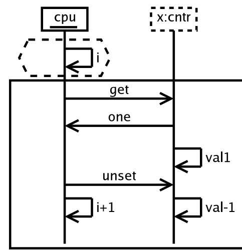


Fig. 18. `dec 1`

Testing whether $c_0 = 0$ is illustrated by Fig. 19. The CPU retrieves the value of *all cntr* and checks that it is indeed either -1 or 1, i.e. nonzero. The encoding of $c_0 \neq 0$ is presented in Fig. 20. CPU simply finds one agent the value of which is 0. Thus, $c_0 \neq 0$, clearly.

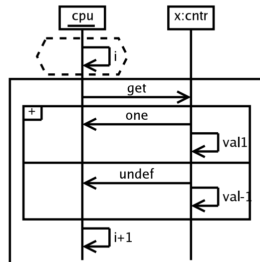


Fig. 19. `zero 0`

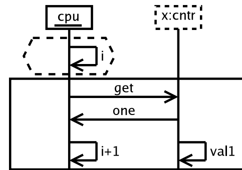


Fig. 20. `not 0`

Finally, in Fig. 21, the LSC imposes that CPU executes `init` infinitely often.

Thus,

(1) all models of the specification execute `halt` infinitely often (Fig. 21);
(2) all models of the specification simulate the 2CM.

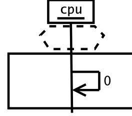


Fig. 21. `halt` infinitely often

□

6 Summary and Discussion

Table 1
Summary of Problem Complexities

Problem	is complete for ...
CCMC	co-NP
OCMC	PSPACE
CDMC	PSPACE
ODMC	PSPACE
Reachability	PSPACE
LSC-Impl	PSPACE
LSC-SAT	PSPACE
CCAD	PSPACE
COAD	EXPTIME
LTL-Cons-COAD	2EXPTIME
DOAD	(Undecidable)
SymLSC-SAT	(Undecidable)

Table 1 summarizes our complexity results. There are two axes along which complexity increases. The distributed version of the problems is always harder than the centralized one, as in [43], while synthesis is also more complex than model checking, for it adds alternation to the problem [50].

The most interesting part is to investigate what causes such a high complexity. We identify two factors making LSCs complex.

(1) LSC semantics relies on partial orders. We used this in the proof of co-NP-completeness of COAD (Th.12) and the lower-bound on the size of synthesized state machines (Th.26). With a chart of size n, we can thus encode a set of runs of size $2^{O(n)}$.

(2) An LSC specification is unstructured. In the PSPACE-hardness proofs, we used LSCs of constant size only and, actually, very short ones, in which events were linearly ordered. The complexity of the specification comes from the fact that many LSCs are active at the same time, describing concurrent liveness properties.

The former cause of complexity is often avoided in practice, because real-world specifications tend to make use of almost linearly ordered scenarios. The latter cause is more difficult to deal with. One shall find ways to describe the problem structure in these models and, more importantly, to rely on this additional information to get more efficient algorithms [53]. This is all but an easy task, as it contradicts one of the basic principles of scenario-based software engineering: requirements are partial, redundant, complementary and range over several aspects of the system. And indeed, agents typically pursue several goals and obey several protocols at the same time.

Undecidability of distributed synthesis means that we need to find other ways to cope with that problem. In [11], we propose such an algorithm, which is sound but not complete. It applies a predefined “implementation scheme” and then checks whether the distributed implementation obtained is correct. We still have to extend it with knowledge and intention to make it applicable to agents.

Finally, we need to investigate further the complexity of loops and we shall try to find a polynomial translation to LTL which resists to the language extensions presented in Sec.5.

References

[1] Amyot, D., Eberlein, A.: An evaluation of scenarion notations for telecommunication systems development. Telecommunications Systems Journal **24** (2003) 61–94

[2] International Telecommunication Union (ITU) Geneva: MSC-2000: ITU-T Recommendation Z.120 : Message Sequence Chart (MSC). (2000) `http://www.itu.int/`.

[3] Object Management Group (UML Revision Task Force): OMG UML Specification (2.0). (2003) `http://www.omg.org/uml`.

[4] Huget, M.P.: FIPA Modeling: Interaction Diagrams. Foundation for Intelliget Physical Agent, Geneva, Switzerland. (2003) `http://www.auml.org/auml/documents/`.

[5] Bontemps, Y., Schobbens, P.Y.: On the semantics of uml 2.0 interaction diagrams. Technical report, University of Namur (2004) Contact the author

to obtain a copy of this paper.

[6] Damm, W., Harel, D.: LSCs: Breathing life into message sequence charts. Formal Methods in System Design **19** (2001) 45–80

[7] Harel, D., Kugler, H.: Synthesizing State-Based Object Systems from LSC Specifications. International Journal of Foundations of Computer Science **13** (2002) 5–51 (Preliminary version appeared in, *Proc. Fifth Int. Conf. on Implementation and Application of Automata (CIAA 2000)*, LNCS 2088, July 2000.).

[8] Bontemps, Y.: Automated Verification of State-based Specifications Against Scenarios (A Step towards Relating Inter-Object to Intra-Object Specifications). Master's thesis, University of Namur, rue Grandgagnage, 21 - 5000 Namur(Belgium) (2001)

[9] Bontemps, Y., Schobbens, P.Y., Löding, C.: Synthesis of open reactive systems from scenario-based specifications. Fundamenta Informaticae **62** (2004) 139–169

[10] Peled, D.A., Clarke, E.M., Grumberg, O.: Model Checking. MIT Press, Cambridge, Massachusetts (2000) ISBN 02-620327-08.

[11] Bontemps, Y., Heymans, P.: As fast as sound (lightweight formal scenario synthesis and verification). In Giese, H., Krüger, I., eds.: Proc. of the 3rd Int. Workshop on "Scenarios and State Machines: Models, Algorithms and Tools" (SCESM'04), Edinburgh, IEE (2004) 27–34 available at `http://www.info.fundp.ac.be/~ybo`.

[12] Dunne, P., Laurence, M., Wooldridge, M.: Complexity results for agent design problems. Annals of Mathematics, Computing and Teleinformatics **1** (2003) 19–36

[13] Wooldridge, M., Dunne, P.E.: The computational complexity of agent verification. In Meyer, J.J.C., Tambe, M., eds.: Agents VIII (Proc. of 8th International ATAL Workshop 2001. Volume 2333 of Lect. Notes in Artificial Intelligence., Springer (2002) 115–127

[14] Walton, C.D.: Multi-agent dialogue protocols. In: Proceedings of the 8th International Symposium on Artificial Intelligence and Mathematics, Ft. Lauderdale, FL (2004)

[15] Harel, D.: Statecharts: a Visual Formalism for Complex Systems. Science of Computer Programming **8** (1987) 231–274

[16] Esteva, M., Rodrìguez, J., Sierra, C., Garcia, P., Arcos, J.: On the formal specification of electronic institutions. In: Agent-Mediated Electronic Commerce (The European AgentLink Perspective). Volume 1991 of Lect. Notes in Artificial Intelligence., Springer (2001) 126–147

[17] Holzmann, G.J.: The model checker SPIN. Software Engineering **23** (1997) 279–295

[18] Rao, A.S.: Agentspeak(l): Bdi agents speak out in a logical computable language. In: Proceedings of the 7th European workshop on Modelling autonomous agents in a multi-agent world : agents breaking away, Springer-Verlag New York, Inc. (1996) 42–55

[19] Hindriks, K.V., Boer, F.S.D., Hoek, W.V.D., Meyer, J.J.C.: Agent programming in 3apl. Autonomous Agents and Multi-Agent Systems **2** (1999) 357–401

[20] De Giacomo, G., Lespérance, Y., Levesque, H.J.: Congolog, a concurrent programming language based on the situation calculus. Artificial Intelligence **121** (2000) 109–169

[21] Bordini, R.H., Fisher, M., Pardavila, C., Wooldridge, M.: Model checking agent speak. In Rosenschein, J.S., Sandholm, T., Wooldridge, M., Yokoo, M., eds.: Proceedings of the Second International Joint Conference on Autonomous Agents and Multi-Agent Systems (AAMAS-2003), Melbourne, Australia, ACM Press (2003) 409–416

[22] Wooldridge, M., Fisher, M., Huget, M.P., Parsons, S.: Model checking multi-agent systems with mable. In: Proceedings of the first international joint conference on Autonomous agents and multiagent systems, ACM Press (2002) 952–959

[23] Foundations for Intelligent and Physical Agents: Specification part 2 - Agent Communication Language. (1999) `http://www.fipa.org`.

[24] Huget, M.P., Wooldridge, M.: Model checking for acl compliance verification. In Rosenschein, J.S., Sandholm, T., Wooldridge, M., Yokoo, M., eds.: Proceedings of the Second International Joint Conference on Autonomous Agents and Multi-Agent Systems (AAMAS-2003), Melbourne, Australia, ACM Press (2003)

[25] Muscholl, A., Peled, D., Su, Z.: Deciding Properties of Message Sequence Charts. Foundations of Software Science and Computer Structures (1998)

[26] FIPA: Case Study of Agent-Oriented Modelling (The UN Security Council's Procedure to Issue Resolutions). (2003) `http://www.auml.org`.

[27] Marelly, R., Harel, D., Kugler, H.: Multiple Instances and Symbolic Variables in Executable Sequence Charts. In: Proc. 17th Ann. ACM Conf. on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'02), Seattle, WA (2002) 83–100

[28] Bontemps, Y., Heymans, P., Kugler, H.: Applying LSCs to the specification of an air traffic control system. In Uchitel, S., Bordeleau, F., eds.: Proc. of the 2nd Int. Workshop on "Scenarios and State Machines: Models, Algorithms and Tools" (SCESM'03), at the 25th Int. Conf. on Soft. Eng. (ICSE'03), Portland, OR, USA, IEEE (2003) available at `http://www.info.fundp.ac.be/~ybo`.

[29] Bohn, J., Damm, W., Klose, J., Moik, A., Wittke, H.: Modeling and validating train system applications using statemate and live sequence charts. In Ehrig, H., Krämer, B.J., Ertas, A., eds.: Proceedings of the Conference on Integrated Design and Process Technology (IDPT2002), Society for Design and Process Science (2002)

[30] Bunker, A., Gopalakrishnan, G.: Verifying a VCI Bus Interface Model Using an LSC-based Specification. In Ehrig, H., Krämer, B.J., Ertas, A., eds.: Proceedings of the Sixth Biennial World Conference on Integrated Design and Process Technology, Society of Design and Process Science (2002) 48

[31] Kam, N., Harel, D., Kugler, H., Marelly, R., Pnueli, A., Hubbard, J.A., Stern, M.J.: Formal modelling of c. elegans development; a scenario-based approach. In Ciobanu, G., Rozenberg, G., eds.: Modelling in Molecular Biology. Natural Computing Series. Springer (2004) 151–173

[32] Wolper, P.: Temporal logic can be more expressive. Information and Control **56** (1983)

[33] Thomas, W.: Languages, automata, and logic. In Rozenberg, G., Salomaa, A., eds.: Handbook of formal languages. Volume 3. Springer, Berlin, New-York (1997) ISBN 3-540-61486-9.

[34] Thomas, W.: Automata on infinite objects. In van Leeuwen, J., ed.: Handbook of Theoretical Computer Science. Volume B. MIT Press and Elsevier Science Publishers, Amsterdam, The Netherlands and Cambridge, Massachussets (1990) 134–191 ISBN 0-262-72015-9 (Second Printing, 1998).

[35] Maidl, M.: The common fragment of CTL and LTL. In: Proc. 41st Annual Symposium on Foundations of Computer Science. (2000) 643–652

[36] Bontemps, Y.: Realizability of scenario-based specifications. Diplôme d'études approfondies, Facultés Universitaires Notre-Dame de la Paix, Institut d'Informatique (University of Namur, Computer Science Dept), rue Grandgagnage, 21, B5000 - Namur (Belgium) (2003)

[37] Kugler, H., Harel, D., Pnueli, A., Yuan, L., Bontemps, Y.: Temporal Logic for Live Sequence Charts. Unpublished draft (2001)

[38] Sistla, A.P., Clarke, E.M.: Complexity of propositional temporal logics. Journal of the ACM **32** (1985) 733–749

[39] Vardi, Y., M., Wolper, P.: Automata-theoretic techniques for modal logics of programs. Journal of Computer and System Science **32** (1986) 183–221

[40] Russell, S.J., Subramanian, D.: Provably bounded-optimal agents. Journal of Artificial Intelligence Research **2** (1995) 575–609

[41] Lynch, N.A., Tuttle, M.R.: An introduction to input/output automata. CWI Quarterly **2** (1989) 219–246

[42] Papadimitriou, C.H.: Computational Complexity. Addison-Wesley (1994)

[43] Harel, D., Vardi, M.Y., Kupferman, O.: On the complexity of verifying concurrent transition systems. Information and Computation **173** (2002)

[44] Savitch, W.J.: Relationships between nondeterministic and deterministic tape complexities. Journal of Computer and System Sciences **4** (1970) 177–192

[45] Harel, D., Marelly, R.: Come, let's play! Scenario-based programming using LSCs and the Play-engine. Springer (2003) ISBN 3-540-00787-3.

[46] Harel, D.: From play-in scenarios to code: An achievable dream. IEEE Computer **34** (2001) 53–60

[47] Abadi, M., Lamport, L., Wolper, P.: Realizable and unrealizable specifications of reactive systems. In Ausiello, G., Dezani-Ciancaglini, M., Rocca, S.R.D., eds.: Automata, Languages and Programming, 16th International Colloquium, ICALP89, Stresa, Italy, July 11-15, 1989, Proceedings. Volume 372 of Lect. Notes in Comp. Sci., Springer (1989)

[48] Kirsten, D.: Alternating tree automata and parity games. In Grädel, E., Thomas, W., Wilke, T., eds.: Automata Logics, and Infinite Games: A Guide to Current Research. Volume 2500 of Lect. Notes in Comp. Sci. Springer (2002) 385 ISBN 3-540-00388-6.

[49] Pnueli, A., Rosner, R.: On the Synthesis of a Reactive Module. In: Proceedings of the sixteenth annual ACM symposium on Principles of programming languages. (1989) 179–190

[50] Chandra, Ashok K.and Kozen, D.C., Stockmeyer, L.J.: Alternation. Journal of the ACM **28** (1981) 114–133

[51] Pnueli, A., Rosner, R.: On the synthesis of an asynchronous reactive module. In Ausiello, G., Dezani-Ciancaglini, M., Rocca, S.R.D., eds.: Automata, Languages and Programming, 16th International Colloquium (ICALP). Volume 372 of Lect. Notes in Comp. Sci., Stresa, Italy, Springer-Verlag (1989) 652–671

[52] Tripakis, J.: Undecidable problems of decentralized observation and control on regular languages. Information Processing Letters **90** (2004)

[53] Alur, R., Grosu, R., McDougall, M.: Modular Refinement of Hierarchic Reactive Machines. In: Proc. of POPL'00, the 27th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Boston, Massachusetts, ACM SIGPLAN-SIGACT (2000)