The computational complexity of scenario-based agent verification and design

Yves Bontemps ¹, Pierre-Yves Schobbens *

University of Namur, Institut d’Informatique, Belgium
Available online 5 April 2006

Abstract

We first advocate that the AUML (Agent Unified Modeling 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 modeling 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 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.

© 2005 Elsevier B.V. All rights reserved.

Keywords: 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 motivated by goals. In order to achieve their goals, which are realized through plans, agents need to coordinate. This coordination does not “just happen”. It has to follow well-specified protocols. Thus, on the one hand, agents must have well-defined plans, but, on the other hand, their behavior, induced by these plans, must comply with interaction 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. 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 (MSC), which are standardized by the International Telecommunication Union (ITU), are by far the most popular of these languages [32]. They present, in an intuitive way, how processes interact, through message passing. This language has been incorporated in the UML, as “Interaction Diagrams” [42]. In the agent world, the Foundations of Intelligent Physical Agents (FIPA) is defining a unified language, based on the Unified Modeling Language (UML), called Agent Unified Modeling Language (AUML) [29], for modeling agent systems. However, this language also inherits the problems that are found both in the UML and in ITU languages. First, UML 2.0 only partially specifies the semantics of Interaction Diagrams, which opens the way to ambiguities [12,16]. Second, MSCs themselves, 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. 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 (LSC) [18]. This language extends exactly MSCs (and Interaction Diagrams) with those syntactic constructs. Hence, one can distinguish between optional and mandatory behavior.

Actually, Live Sequence Charts provide engineers with a graphical front-end to Temporal Logic [10,26]. However, this language remains (i) graphical and (ii) scenario-based. In [14], 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 [44], whereas agent design is classically referred to as synthesis [3,45]. In this paper, we use the term “agent design”, following Dunne and Wooldridge [19,59], and Stewart [51].

Here, we show that many simple problems on (non-hierarchical) LSC have a surprisingly high complexity, and specially that the automated synthesis of a distributed agent system, is undecidable. This may seem to render our dream unachievable, but actually it is hardly surprising that distributed software development, that requires the brains of millions of programmers worldwide and in which still today unexpected bugs are found, is undecidable. This means that more knowledge has to be put in the synthesis algorithms, e.g. as heuristics [7]. Thus although the dream will never be fully achieved, we can try to come close enough to it to alleviate the work of programmers of distributed systems.

The work of Wooldridge and colleagues is related to what we present here [19,59]. 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 \( \Sigma^u_P \) (i.e. recognizable in polynomial time by a Turing Machine with \( u \) calls to an NP oracle), the complexity of verification is \( \Pi_{u+1}^P \), i.e. exactly one universal alternation is introduced. When the task description is PSPACE-complete, verification is PSPACE-complete as well. Stewart extends and sharpens those results, by studying achievement and maintenance agent design problems in bounded problems, when these problems are parameterized by the number of environment states and the number of agent actions [51]. Action effects are history-dependent and specified by a polynomial-time Turing Machine. Stewart show that, in this setting, even with very low number of states and actions, resp. three and two, achievement is PSPACE-complete. The same is true of the maintenance problem.

In computer science, the synthesis problem was originally posed by Church [15] and solved by Büchi and Landweber [9]. This problem became popular years later and has been studied, in the framework of Temporal Logic, by Pnueli and Rosner [45]. They showed that the problem was 2EXPTIME-complete. This high complexity comes from two problems: first, the tableau automaton built from an LTL formula may be exponentially larger than the original it is built from and, secondly, this nondeterministic automaton needs to be determinized to solve the synthesis problem. If the tableau automaton is directly deterministic, an exponential is saved [47]. We benefit from this fact in our procedure: a simply-exponential deterministic Büchi automaton can be built from an LSC specification. The synthesis problem was later revisited by Vardi [55], extending previous results to CTL*. Pnueli and Rosner also showed that the problem of synthesizing a distributed program over arbitrary architectures was undecidable [46]. They also propose
a sufficient criterion that makes distributed synthesis not harder than synthesis: the architecture must be adequately connected. Roughly, an architecture is adequately connected if every process in charge of producing outputs can be informed of all received inputs. Finally, they consider the problem of synthesis over pipeline and hierarchies of adequately connected architectures. They show it to be nonelementary decidable. These results have been later extended by Kupferman and Vardi [36], taking into account cyclic architectures as well. Madhusudhan and Thiagarajan study the problem of distributed synthesis and give three criteria for claiming decidability: trace-closure of the specification, implementations must be clocked, i.e. reactions are not history-dependent but only depend on the current time, and communication rigid, i.e. at any stage in the computation, a process may not synchronize with more than one other process [41]. These restrictions are strong but already, they hint to one possible cause of undecidability in LSCs: they are not trace-closed. Gastin et al. consider asynchronous games played over traces and obtain decidability results for recognizable trace languages, when allowing players to use causal memory [23]. Again, these decision procedures are nonelementary.

Our work is strongly related to these approaches. We consider the same problem and obtain the same results: synthesis is difficult and distributed synthesis is undecidable. This paper brings two main contributions. First, it considers a “natural” language, built by and for practitioners. We therefore show the practical relevance of these theoretical results. Second, it considers a language which is much less expressive than temporal logics (LTL, CTL, ACTL\textsuperscript{det}) or automata [13]. In some sense, our intractability results are therefore “sharper” than those presented above.

Even though LSC are less expressive than common temporal logic, it is incomparable to other restricted subsets of LTL. We show in [13] that LSCs are not closed under any boolean operation and that LSC specifications are only closed under intersection. In comparison, Emerson studies the complexity of satisfiability and model checking of LTL, with temporal operators restricted to $\Diamond$ and $\Box$ [20]. LSC-definable languages are incomparable to languages definable in this fragment of LTL, because LSCs make a very particular use of both $\Box$ and $\Diamond$ operators. Our results are therefore related to, but do not follow from, those of Emerson.

Walton presents a lightweight language for describing agent dialogues, named Multi-Agent Protocol [58]. This language is based on the theory of Speech Act and is intended to be an alternative to Statecharts [24], which are used in Electronic Institutions [21]. Walton proposes a translation of MAP to PROMELA, the input language of the SPIN model-checker [28], 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 purposely abstract on the actual form of agents implementing these protocols. MAP could be such an implementation language.

Another possibility would be to use agent-oriented programming languages, such as AgentSpeak, for instance. There is also some tool support for the verification of AgentSpeak programs [5]. 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 [60]. 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, for describing agent protocols, is not as rich as other agent communication languages, such as FIPA’s ACL [22]. 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 [31].

Nevertheless, LSCs are a flavor of MSCs, which stemmed from and have been widely used in 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 AULM. They will probably have to be tuned to support features from agent-oriented programming paradigm, most notably speech act communication. Nevertheless, this paper shows that, even in the absence of such fancy constructs, this extremely simple language gives rise to problems that are already intractable or undecidable.
The paper is structured as follows. We present, in Section 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 state-based formalism, here input/output automata, encoding strategies, as presented in Section 2.2. This section concludes by defining when a design model is a correct implementation of a scenario-based specification. In Section 3, verification problems are considered. First, checking whether a design model is a correct implementation (Section 3.1) and then, whether a specification refines another specification (Section 3.2). The question of whether a specification is implementable is investigated in Section 4. Section 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 Section 6, we summarize the results and put them in perspective.

2. Models

We assume that we are given a finite set of agent names $A_g$ and of message names $M$. An event is a triple from $A_g \times M \times A_g$. The set of events is $\Sigma$. We will denote the set of events “sent”, or triggered, (resp. “received”, or sensed) by some agent $a$ with $\Sigma_{s}^a$ (resp. $\Sigma_{r}^a$) and let $\Sigma_a = \Sigma_{s}^a \cup \Sigma_{r}^a$. An event of the form $(a_1, m, a_2)$ represents the fact that $a_1$ sends message $m$ to $a_2$. $\Sigma^*$ represents the set of all finite sequences of events, while $\Sigma^\omega$ 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 $\Sigma'$.

We assume here, for simplicity, that communication is instantaneous. (In contrast, some undecidability proofs of [40] 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 (LSC) [18] are based on Message Sequence Charts (MSCs) [32]. They present the various interactions of agents. Every agent owns a “life-line”, labeled by its name, e.g. “agenda” in Fig. 2. 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. This case study is concerned with the modeling of the voting protocol followed by the United Nation Security Council for issuing resolutions. It is unclear whether the scenario of Fig. 1 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 [18]. 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).

Marelly, Harel and Kugler have extended LSC with symbolic instances [39]. 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.

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. Note 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 between events. The underlying idea is that two events involving different agents cannot be ordered, as they take place at separate places “across the network”. Precisely, the temporal ordering of events is the transitive closure of the relation defined by the following three clauses: (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. For example, 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.

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) [8], a radio-based train system [4], virtual wrappers for PCI bus [6] and some part of the C elegans worm [33]. Examples displayed in Figs. 4, 5, 6 and 7 are based on the CTAS system. The CTAS system combines several processes to offer automated services, to support air traffic
controllers. The system is made of a central communication manager “cm” to which processes (“clients”) connect. Some computations performed by these processes depend heavily on accurate weather reports and, more importantly, these processes must absolutely use consistent weather data. Otherwise, conflicting results could arise. The protocol described by the LSCs of Figs. 4, 5, 6 and 7 has been designed to ensure that all clients are synchronized on the data they use. When new data is available, all clients are asked to retrieve it. If some client fails to fetch the new report, the system tries to roll back to the previous version.

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.

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 \( \Sigma' \)-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 (\( \gamma \in \Sigma^\omega \)). An interpretation satisfies an LPO if its restriction to \( \Sigma' \) 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 (|= \Sigma^\omega \times \text{LPO}).** \( \gamma \models \langle L, \leq, \lambda, \Sigma' \rangle \) iff

- \( \gamma \in \Sigma^* \)
- \( \gamma|_{\Sigma'} \) 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 $\Sigma_R$, is a couple of $\Sigma_R$-LPOs $\Box(P,M)$. We refer to $P$ as the *prechart* and to $M$ as the *main chart*. We remark that $P$ and $M$ must be defined over the same alphabet, that we let be $\Sigma_R$.

- A *existential LSC* (eLSC), with restricted events $\Sigma_R$, is a $\Sigma_R$-LPO $\Diamond(M)$.

The size of an LSC is its number of locations. 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 $\gamma$, $M$ is also satisfied immediately after.

**Definition 4 ($|= \subseteq \Sigma^\omega \times \text{LSC}$).** Let $\gamma \in \Sigma^\omega$.

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

- $\gamma |= \Diamond(M)$ iff $\exists u \in \Sigma^*, \gamma' \in \Sigma^\omega : \gamma = u\gamma' \text{ and } \gamma' |= M$.

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

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

- $L |= \Box(P,M)$ iff for every $\gamma \in L$, $\gamma |= \Box(P,M)$.

- $L |= \Diamond(M)$ iff there is some $\gamma \in L$ such that $\gamma |= \Diamond(M)$.

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 $S$ is a finite set of uLSCs. Its model relation is defined as the conjunction of the model relation of its members: $\gamma |= S$ iff for every $U \in S$, $\gamma |= U$.

The size of a specification is the sum of the size of its members. A language $L$ is defined by an LSC $S$ if $L |= S$.

A classical question, with respect to logics, is their relation to classes of languages, usually via automata [53,61]. 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 $\omega$-regular languages and a very restricted sub-class indeed. Live Sequence Charts are strictly less expressive than Deterministic Büchi Automata (DBA) [52] and ACTL$^{\text{det}}$, the common fragment of LTL and ACTL [38], as we showed in [111]. 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 [10,26]. Another polynomial translation had already been proposed by Kugler et al. [34]. Yet, their translation applies only to LSCs in which no event appears twice.

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

$$\forall \gamma \in \Sigma^\omega : \gamma |= \phi_S \iff \gamma |= S.$$ 

**Proof.** We just show how to translate a single uLSC $L$ to an equivalent LTL formula $\phi_L$ of size $O(|L|^5)$. The overall formula is just the conjunction $\bigwedge_{L \in S} \phi_L$. Let $L = \Box(P,M)$. 

We define the index of a location $l$ in an LPO as $\overline{idx}(l) = |\{l' \mid \lambda(l) = \lambda(l') \land l' \leq l\}|$. 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 $(\overline{idx}(l), \lambda(l))$ 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

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

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

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

where notoccurs($l$) asserts that there will not be $\overline{idx}(l)$ occurrences of $\lambda(l)$ before having seen $|P|$ occurrences of restricted ($\Sigma_R$) 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 $\overline{idx}(l')$ when the $\overline{idx}(l)$-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 $\overline{idx}(l)$ and $\overline{idx}(l')$ 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 [50], 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-formulæ [57].

Every LSC specification is equivalent to the conjunction of liveness and safety properties, one for every event in $\Sigma$ [14]. A scenario $S$, with restricted events $\Sigma_R$, forbids $e \in \Sigma$ after a finite run $w \in \Sigma^*$ if 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 $I \supset P$ 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 $\Sigma_R$ 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 $M : I \subset M \land t \models I$,
  - $\forall I'$ : 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 $M : I \subset M \land t \models I$,
  - $\exists I'$ : ideal of $M : we \models I'$.

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 $\gamma$. It is $e$-live iff for every prefix $w$ of $\gamma$, if some scenario requires $e$ after $w$, then $e$ eventually occurs after $w$. 
The following theorem has been shown in [14] 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 = ΣR-live ∧ ΣR-safe). For every γ ∈ Σω,

\[ γ \models \Box (P, M) \iff \forall e ∈ Σ : γ \text{ is } e\text{-safe and } e\text{-live}, \text{ wrt } \Box (P, M). \]

### 2.2. Strategies

Agents are partitioned into two teams: the environment and the system. Formally, \( Ag = Sys \cup Env \). System-controlled events are \( Σ_{Sys} = Sys \times M \times Ag \). 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 [48,59]. Remember that we abstract away from agent’s actions and focus on coordination instead. Thus, our abstract view of agent \( a \) is a strategy \( f : Σ^* → 2^{Σ_A^l} \). 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 [37]. An input-output automaton for agent \( a ∈ Ag \) is a finite automaton the alphabet of which is \( Σ_a \). A distinction is made between input events \( (Σ_r) \) and output events \( (Σ_o) \). 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 \( Σ_A^l \) 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 \( A \), denoted \( L(A) \), is the set of words generated by \( A \)'s fair runs. The composition of two I/O automata \( (A_1 × A_2) \) is defined as the synchronous product of \( A_1 \) and \( A_2 \), see [37] 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 \( f : Σ^* → 2^{Σ_A^l} \). It is of finite memory if there is an equivalence relation \( ≃ \) on \( Σ^* \) such that (1) \( \forall i \geq 0 : u_i ≃ u_i' \) implies \( f(u_i) = f(u_i') \). 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 \( Σ_A^l \) events appear only according to the strategy:

\[ Out(f) = \{ u_{0e0}u_{1e1}... | \forall i \geq 0 : u_i ∈ (Σ \setminus Σ_A^l)^* \text{ and } e_i ∈ f(u_{0e0}...u_i) \}. \]

Agents can be organized in societies. A society is a set of agents \( A ⊆ Ag \). Its triggered events and sensed events are the union of all triggered/sensed events of its composing agents: \( Σ_A^t = \bigcup_{a ∈ A} Σ_a^t \) and \( Σ_A^s = \bigcup_{a ∈ A} Σ_a^s \). The strategy of \( A \) is also the union of its agent’s strategies: \( f_A(w) = \bigcup_{a ∈ A} f_a(w) \).

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

\[ ∀γ ∈ Out(f_{Sys}) : \begin{cases} γ \text{ is } Σ_{Env}\text{-live } \implies γ \text{ is } Σ_{Sys}\text{-live}, \\ γ \text{ is } Σ_{Env}\text{-safe } \implies γ \text{ is } Σ_{Sys}\text{-safe}. \end{cases} \]
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 $A$, and an LSC specification $S$, decide whether $Out(f_{Ag}) \models S$.”

**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 [43], 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 $A$ and a simple path in the synchronous product $A \times A_L$, with $A_L$ is the linear nondeterministic Büchi automaton recognizing all counter-examples of $L$. Note that the simple path is at most of length $n \times |A|$. $\square$

**Proof.** (Hardness) There is a polynomial reduction of COMPLEMENT TSP (see [43]) 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 $\langle v, n \rangle$, there is a transition $\langle v, n \rangle \xrightarrow{v'} \langle v', n + d_{qq'} \rangle$, iff the edge between $q$ and $q'$ in $G$ has weight $d_{qq'}$. Thus, a path $\langle v_0, i_0 \rangle \ldots \langle v_j, i_j \rangle$ in the automaton corresponds to a path $v_0 \ldots v_j$ in $G$. Furthermore, the total weight of $v_0 \ldots v_j$ is $i_j$.

In any state, there is also a transition to the “down-counting” states: $\langle v, n \rangle \xrightarrow{\$} \langle \$, n \rangle$. From these states, the automaton counts down, decreasing the counter by one unit at a time, until its counter equals 0: for $n > 0$, $\langle \$, n \rangle \xrightarrow{\text{tick}} \langle \$, n - 1 \rangle$. When zero is reached, the automaton reads an infinite sequence of “end” events: $\langle \$, 0 \rangle \xrightarrow{\text{end}} \langle \$, 0 \rangle$. 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 the LSC of Fig. 8: the prechart contains $\{q_1, \ldots, q_n, \$\}$, where $q_1$’s are unordered, whereas $\$ is greater than all $q_i$.

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 $k$. □

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 $A_g$ into $Sys$ and $Env$, a strategy $f_{Sys}$, represented by $A$, and an LSC specification $S$, decide whether $f_{Sys}$ is a correct implementation of $S$ (see Definition 10).”

**Theorem 15.** OCMC is complete for PSPACE.

The proof of this theorem is similar to the proof to be provided in Section 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 distributed 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 Section 2.2.

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

Unfortunately, as usual in verification [30], distribution makes model checking more complex. Now, the problem becomes PSPACE-complete instead of coNP-complete. 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 implementation violates it. This problem is exactly the complement of CDMC, which is thus in coPSPACE = PSPACE, by Savitch’s theorem [49].

**Theorem 17.** CDMC is PSPACE-complete.

**Proof.** *(Membership)* Let $m$ be the size of $A_i$’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 [50]. 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 \( \Gamma \)
and its symbols are \( \Sigma \). 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 [43]. For every cell tape, we build an
automaton, say \( A_i \). The alphabet of the system is \( \{ \text{init}, \text{halt} \} \cup (\Gamma \times \{1, \ldots, n\}) \). An event \((\gamma, i)\)
means that the tape head moves to cell \( i \) and the control location becomes \( \gamma \). \( A_i \) 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 \( \gamma \) 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
\( \gamma' \), of the Turing machine. Let the tape head be on cell \( i \). Then, \( A_i \) will contain a transition \((a, \gamma), (\gamma', i + 1), a')\),
while \( A_{i+1} \) 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, reachabil-
ity in networks of automata is already a difficult problem [43], as hinted to by Theorem 17. The next section presents
simple analysis problems, on LSCs only, that are also difficult. This can be counter-intuitive, as one is naturally led
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 \( S \), decide whether
\[ \mathcal{L}(S) \models L. \]

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 ap-
proach. Given a certain abstract specification \( S \), a more precise specification \( S' \) is designed and we want to verify that
every behavior induced by \( S' \) is a legal behavior of \( S \). Logically, this boils down to verifying the validity of \( S' \rightarrow S \).

Problem 20 (LSC-IMPL). The problem of implication of LSC specifications (LSC-IMPL) is given two LSC specifications \( S \) and \( S' \), to decide whether
\[ \forall \gamma \in \Sigma^\omega : \gamma \models S \implies \gamma \models S'. \]

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 \( S \), whether
\[ \exists \gamma \in \Sigma^\omega : \gamma \models S. \]
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 \( \Phi_u \). The existential LSC can be turned into a “never claim” LTL formula, claiming that the existential LSC is never matched, that we denote \( \phi_e \). Then, we ask whether the formula \( \Phi_u \rightarrow \phi_e \) 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 [50]. 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 finite set \( \Gamma \). 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 \( \Gamma \), the halting location is \( \gamma_0 \), 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:

\[
(\Gamma \cup \{in, $\} \cup \{0, 1\}) \times \{0, \ldots, n\}.
\]

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

\[
init = (in, 0)(0, 0)(in, 1)(0, 1) \ldots (in, j)(0, j) \ldots (in, n)(0, n)(\gamma_0, 0)
\]

The “init” sequence ensures that, at the beginning of the run, the tape cell contains \( n \) blank cells and the initial location is \( \gamma_0 \), 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, \( \gamma \) is the control location and \( i \) is the tape head position. We say that it is encoded by a word \( w \) if

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

Fig. 9. Initialization sequence of DPSPACE TM.
Notice that, when $w$ fulfills these conditions, $C$ can be unambiguously retrieved from $w$.

It is easy to check that $\text{init}$ 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, without loss of generality, that $C = (T, \gamma, i)$, $T[i] = 0$ and $C' = (T', \gamma', i + 1)$, where $T'$ is the result of writing 1 at the $i$th position in $T$. Assume that $C$ is encoded by some word $w$. Then, $w$ is of the form $v \cdot (\gamma, i)$ and the last occurrence of either $\{(0, i), (1, i)\}$ in $v$ is $(0, i)$. The transition to $C'$ will be encoded by building $C'$ as

$$w' = v(\gamma, i)(0, i)(\$, i)(1, i)(\gamma, i + 1).$$

One can check that $w'$ is indeed an encoding of $C'$:

1. it ends with $(\gamma, i + 1)$, the new control location and tape head position;
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. This corresponds exactly to the transition, that left all tape cells unchanged, but the $i$th one.
3. in $w'$, $(1, i)$ indicates that the $i$th cell now contains 1.

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 Figs. 9–12. The first one retrieves the last occurrence of an event of the form $(0, i)$ or $(1, i)$. It is copied immediately after $(\gamma, i)$. This retrieval is presented in Fig. 10. One should take care of a detail, here: we want to be sure that after $(\gamma, i)$, 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.

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.

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 [27]. 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 DSPACE Turing Machine.

![Fig. 10. Retrieving tape cell content.](image-url)
4. Agent design

In this section, we turn to the most complex class of problems considered in this paper: agent design. In computer science, this problem is called synthesis, but we follow the literature about agents and use the term “agent design” [19, 51]. 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” [25].

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 $A$. 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.

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 $L(S)$ 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 [3]. 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 Open Agent Design) is the problem of deciding, given an LSC specification $S$ 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 [14], 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 [35].

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 [45].

**Theorem 25.** COAD is complete for EXPTIME.

**Proof. (Membership)** The algorithm presented in [14] 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 [26].

**Proof. (Hardness)** We encode an alternating PSPACE Turing machine into an LSC, as we did before (see Theorem 22). The result will follow from the fact that APSPACE = EXPTIME [17]. 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 $(\gamma', i, A)(a, i, A)\delta_1(a, i, A)(\gamma', j, A')$, 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 [14] 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 $\phi_n$.

In this game, $Env$ controls $\{a_1, \ldots, a_n\} = \Sigma_{Env}^s = \Sigma_{Sys}^r$ and $Sys$ controls $\{b_1, \ldots, b_n\} = \Sigma_{Sys}^s = \Sigma_{Env}^r$. $Env$ first presents $Sys$ with a sequence of $n$ symbols. Note 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 $\phi_n$ 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 $\phi_n$ grows only quadratically in $n$.

Now, consider some strategy $f : \Sigma^* \rightarrow \Sigma_{Sys}^s$ winning in this game. If $f$ is a correct implementation, it must have enough memory to remember the order in which $a_1 \ldots a_n$ occurred. Otherwise, there would exist two words $w$ and $w'$ where $w' \neq w$ but $f(w') = f(w)$.

![Fig. 13. LSC specification $\phi_n$.](image-url)
of \((\Sigma_{\text{Env}})^*)\) such that \(\text{elts}(w) = \text{elts}(w') = \Sigma_{\text{Env}}^*\) but \(f\) cannot distinguish between them, i.e. \(w \simeq w'\), and thus \(f(w) = f(w')\) (see Section 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 the order of queries (a’s). Contradiction.

All permutations of \(a_1 \ldots a_n\) 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 [38] that turning LSCs to equivalent ACTL\(^{\text{det}}\) formulae also involves an exponential blow-up. Indeed, for every ACTL\(^{\text{det}}\) formula, there is a nondeterministic Büchi automaton recognizing their complement, which is linear in their size.

In light of these results, a natural question to be asked is whether there exists a syntactic restriction of uLSC that always yields succinct automata. We provide below such a restriction, which is still very drastic. This question is thus not closed and is a topic for further research.

Our syntactic restriction of uLSC only allows one event in the prechart. The prechart contains only environment-controlled events and the main chart, system-controlled events. Finally, the system responses to environment stimuli in a “superstep” approach: the environment may not provide any input while the system is answering.

Strategies for these restricted uLSC are simple. They are made of an initial state \(q_0\), in which every environment-controlled event, say \(e\), is allowed and leads to some state \(q_e\). From this state, a sequence of system events \(w_e\) is performed, leading back to \(q_0\). This sequence of events is picked in such a way that it is consistent with the main charts of all uLSCs guarded by \(e\). The synthesis problem amounts to checking whether a set of LPO allows a common run and building this run as a witness. It is easy to see that this problem can be solved in time polynomial in the size of the LPOs, by iteratively computing compatible ideals in the various LPOs. The I/O automaton is small: it contains \(1 + \Sigma_{\text{Env}} + \Sigma_{\text{Env}} \cdot O(n)\) states, because the response sequence is made of \(O(n)\) events, where \(n\) is the number of locations in the LPO.

Note that, if we allow choice in the LPO, checking whether a set of LPO allows a run is NP-complete. However, the automaton remains small.

The problem of centralized realizability is missing 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 [46].

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

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

Theorem 29. LTL-CONS-COAD is complete for 2EXPTIME.
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 \( \Sigma'_a \) 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 \( S \) and a society of agents \( \text{Sys} \), decide whether there is a list of strategies \( \{ f_a \}_{a \in \text{Sys}} \) one for every system agent, such that

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

**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 [54].

We first recall the definition of PCP. A PCP instance is a list of pairs of words \( (w_1, u_1) , \ldots , (w_n, u_n) \), such that, for all \( i \), \( w_i \neq u_i \) and \( w_i, u_i \in \Theta^* \) (for some finite alphabet \( \Theta \)). A solution to a PCP instance is a finite sequence of indexes \( i_1 \ldots i_m (m \geq 1 \text{ and } 1 \leq i_j \leq n, \text{ for all } j) \) such that, \( w_{i_1} w_{i_2} \ldots w_{i_m} = u_{i_1} u_{i_2} \ldots u_{i_m} \). 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 \( \Theta \cup \{ k_1, \ldots , k_n \} \cup \{ \$ \} \cup \{ 0, 1 \} \cup \{ A_0, A_1 \} \), plus an arbitrary finite number of events that can be exchanged between system agents, say \( \{ s_0, \ldots , s_p \} \). The system is made of two agents: \( a_1 \) and \( a_2 \). The first agent may observe \( \Theta \cup \{ \$ \} \), whereas the second can observe \( \{ k_1, \ldots , k_m, \$ \} \). All these events, but \( \{ A_0, A_1 \} \) and the additional system events \( \{ s_0, \ldots , s_l \} \) 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 \( 1 i_1 u_1 \ldots i_m u_m \$ \), 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. \( \Box \)
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 [13]. This extension does not cause any problem, except 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 [13]. 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 need 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 can be used to model 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. We 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. Below, 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 [39], even though their solution has been tuned for animation, and its formalization may not be as clean as it could be.

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

We also assume that we are given a countable set of first-order variables, $\text{Var}$. $\text{Var}$ and $\text{Ag}$ are distinct. An interpretation of $V \subset \text{Var}$ is a function $\theta : V \rightarrow \text{Ag}$. Message terms are also extended to include first-order variables. We write $\Sigma(V)$, for $V \subset \text{Var}$, to denote the set $(\text{Ag} \cup V) \times \mathcal{M} \times (\text{Ag} \cup V)$. Ground events are events from $\Sigma(\theta)$. 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 $\mathcal{I}$ 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 $x \in \text{Var}$, $R \in \text{Role}$ 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 $\theta$, this yields a ground LPO, as well.

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

- $Pop, \theta, \gamma |= Q$, with $Q \in \text{LPO}$ iff $\gamma |= \theta(Q)$.
- $Pop, \theta, \gamma |= \forall x : R : Q$ iff, for every $a \in \text{Agents}$,
  
  $R \in Pop(a) \implies Pop, \theta \cup \{x \mapsto a\}, \gamma |= Q$.
- $Pop, \theta, \gamma |= \exists x : R : Q$ iff, there is some $a \in \text{Agents}$, such that
  
  $R \in Pop(a)$ and $Pop, \theta \cup \{x \mapsto a\}, \gamma |= Q$.

A Symbolic uLSC (SymLSC) 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 [39].
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. Note that matching can be done according to several variable interpretations, and we take all of them into account.

**Definition 33** ($|= \subseteq P \times I \times \Sigma^\omega \times \text{SymLSC}$). $Pop, \gamma |= \Box(P, M)$ iff, for every first-order variable interpretation $\theta$, for every decomposition $uv\gamma'$ of $\gamma$,

$Pop, \theta, v |= P \implies Pop, \theta, \gamma' |= M$.

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

$Pop, \gamma |= S \iff \forall S \in S : Pop, \gamma |= S$.

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

$\exists \gamma \in \Sigma^\omega : Pop, \gamma |= S$.

**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 $\text{prog}$), that has two integer counters $c_0, c_1$ and uses the following statements:

- $\text{init}$ is an initialization statement, that resets $c_0$ and $c_1$ to 0. There is only one $\text{init}$ statement, located at line 0 of $\text{prog}$.
- $\text{go to } l_1 \text{ or } l_2$, where $l_1$ and $l_2$ are line numbers, with $l_1, l_2 > 0$. Executions must jump (nondeterministically) at line $l_1$ or $l_2$.
- $\text{halt}$ is a halting statement. There is only one $\text{halt}$ statement. Its effect is to make the execution back to line zero, i.e. to the $\text{init}$ statement.
- $\text{inc } i$, 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.
• \texttt{dec} \(i\) decrements \(c_i\) and goes on with the statement at the next program line.
• \texttt{not} \(i\). The execution goes on if \(c_i \neq 0\). Otherwise, the execution stops.
• \texttt{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 \texttt{halt} is undecidable. We remark that, if \(prog\) executes \texttt{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 \texttt{init} is undecidable, too.
   Actually, the same finite execution, from \texttt{init} to \texttt{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, \(\text{Role} = \{\text{cntr}\}\). Every agent playing role \texttt{cntr} can assume three “values”: \(-1\), \(0\) and \(1\). The value of counter \(c_i\) is the number of agents assuming value \(i\). We also use a concrete instance, named “\texttt{cpu}”, which is a “central processing unit” (Fig. 14). It executes sequentially the \(2\text{CM}\) statements as prescribed by \(prog\) and sets the values of \texttt{cntr} agents. Agent \texttt{cntr} can receive four messages: “\texttt{get}”, “\texttt{unset}”, “\texttt{set0}” and “\texttt{set1}”. The first one queries the value currently stored (\(-1\), \(0\) or \(1\), thus). The three last messages set the value, as described by Fig. 15.

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

The CPU sends to itself the line number of the next statement to execute. If line \(i\) is a statement of the form \texttt{inc} 1, this line is translated to the LSC of Fig. 17. In this LSC, some agent in \texttt{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. Note that the execution proceeds at the next line, i.e. \(i + 1\).

The same approach is taken to translate the statement \texttt{dec} 1. This is illustrated by Fig. 18.

Testing whether \(c_0 = 0\) is illustrated by Fig. 19. The CPU retrieves the value of all \texttt{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.

Finally, in Fig. 21, the LSC imposes that CPU executes \texttt{init} infinitely often.

Thus,
Fig. 18. dec 1.

Fig. 19. zero 0.

Fig. 20. not 0.

Fig. 21. halt infinitely often.

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

6. Summary and discussion

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 [30], while synthesis is also more complex than model checking, for it adds alternation to the problem [17].

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 (Theorem 12) and the lower-bound on the size of synthesized state machines (Theorem 26). With a chart of size $n$, we can thus encode a set of runs of size $2^{O(n)}$.

<table>
<thead>
<tr>
<th>Problem</th>
<th>is complete for …</th>
</tr>
</thead>
<tbody>
<tr>
<td>CCMC</td>
<td>co-NP</td>
</tr>
<tr>
<td>OCMC</td>
<td>PSPACE</td>
</tr>
<tr>
<td>CDMC</td>
<td>PSPACE</td>
</tr>
<tr>
<td>ODMC</td>
<td>PSPACE</td>
</tr>
<tr>
<td>REACHABILITY</td>
<td>PSPACE</td>
</tr>
<tr>
<td>LSC-IMPL</td>
<td>PSPACE</td>
</tr>
<tr>
<td>LSC-SAT</td>
<td>PSPACE</td>
</tr>
<tr>
<td>CCAD</td>
<td>PSPACE</td>
</tr>
<tr>
<td>COAD</td>
<td>EXPTIME</td>
</tr>
<tr>
<td>LTL-CONS-COAD</td>
<td>2EXPTIME</td>
</tr>
<tr>
<td>DOAD</td>
<td>(Undecidable)</td>
</tr>
<tr>
<td>SYMLSC-SAT</td>
<td>(Undecidable)</td>
</tr>
</tbody>
</table>
(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 [2]. 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.

Undecidability of distributed synthesis means that we need to find other ways to cope with that problem. In [7], 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.

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

References


