

Reasoning about sequences of memory states^{*}

Rémi Brochenin, Stéphane Demri, and Etienne Lozes

LSV, ENS Cachan, CNRS, INRIA
{brocheni,demri,lozes}@lsv.ens-cachan.fr

Abstract. In order to verify programs with pointer variables, we introduce a temporal logic LTL^{mem} whose underlying assertion language is the quantifier-free fragment of separation logic and the temporal logic on the top of it is the standard linear-time temporal logic LTL. We analyze the complexity of various model-checking and satisfiability problems for LTL^{mem} , considering various fragments of separation logic (including pointer arithmetic), various classes of models (with or without constant heap), and the influence of fixing the initial memory state. We provide a complete picture based on these criteria. Our main decidability result is PSPACE-completeness of the satisfiability problems on the record fragment and on a classical fragment allowing pointer arithmetic. Σ_1^0 -completeness or Σ_1^1 -completeness results are established for various problems by reducing standard problems for Minsky machines, and underline the tightness of our decidability results.

1 Introduction

Verification of programs with pointers. Model-checking of infinite-state systems is a very active area of formal verification [BCMS01] even though in full generality, simple reachability questions are undecidable. Nevertheless, many classes of infinite-state systems can be analyzed, such as Petri nets, timed automata, etc. Programs with pointer variables suffer the same drawback since reachability problems are also undecidable, see e.g. [BFN04,BBH⁺06]. It is worth noting that specific properties need to be verified for such programs, such as the existence of memory leaks, memory violation, or shape analysis. Prominent logics for analyzing such programs are Separation Logic [Rey02], pointer assertion logic PAL [JJKS97], TVLA [LAS00] and alias logic [BIL04], to quote a few examples. *Temporal Separation Logic: what for?* Since [Pnu77], temporal logics are also used as languages for formal specification of programs. General and powerful automata-based techniques for verification have been developed, see e.g. [VW94]. On the other hand, Separation Logic is a static logic for program annotation [Rey02], and more recently for symbolic computation [BCO05]. Extending the scope of application of Separation Logic to standard temporal logic-based verification techniques has many potential interests. First, it provides a rich underlying assertion language where properties more complex than accessibility

^{*} Work supported by the RNTL project “AVERILES”. The first author is supported by a fellowship from CNRS/DGA.

can be stated. Second, this probably yields a significant feedback for the purely static Separation Logic extended with general recursion, which has not been very studied up to now. For instance, if we write Xx to denote the next value of x (also sometimes written x'), the formula $(x \hookrightarrow Xx) \cup (x \hookrightarrow \text{null})$, understood on a model with constant heap, characterises the existence of a simple flat list, which is usually written $\mu L(x). x \hookrightarrow \text{null} \vee \exists x'. x \hookrightarrow x' \wedge L(x')$. Third, temporal logics allow to work in the very convenient framework of "programs-as-formulae" and decision procedures for logical problems can be directly used for program verification. For instance, the previous formula can be seen as a program walking on a list, and more generally programs without destructive updates can be expressed as formulae. Some programs with destructive updates that perform a simple pass on the heap, have an input-output relation that may be described by a formula. For instance, the formula $(x \hookrightarrow_0 Xx \wedge Xx \hookrightarrow_1 x) \cup x \hookrightarrow_0 \text{null}$ expresses broadly that the list in initial heap h_0 is reversed in final heap h_1 . Fourth, pointer arithmetic has been poorly studied until now, whereas arithmetical constraints in temporal logics are known to lead to undecidability, see e.g. [CC00]. Actually, there is a growing interest in understanding the interplay of pointer arithmetic, temporal reasoning, and non aliasing properties.

Our contribution. We introduce a linear-time temporal logic LTL^{mem} to specify sequences of memory states with underlying assertion language based on quantifier-free Separation Logic [Rey02]. From a logical perspective, the logic LTL^{mem} can be viewed as a many-dimensional logic [GKWZ03] since LTL^{mem} contains a temporal dimension and the spatial dimension for memory states. Our logic addresses a very general notion of models, including the aspects of pointer arithmetic and recursive structures with records. We distinguish the satisfiability problems from the model-checking problems, as well as distinct subclasses of interesting programs, like for instance the programs without destructive update. The result that is the most promising for future implementation is the PSPACE-completeness of the satisfiability problems SAT(CL) and SAT(RF) where CL is the classical fragment without separation connectives and RF is the record fragment with no pointer arithmetic but with separation connectives. This result is very tight, as both propositional LTL and static Separation Logic are already PSPACE-complete [SC85,CYO01]. These results are obtained by reduction to the nonemptiness problem for Büchi automata on an alphabet of symbolic memory states obtained by an abstraction that we show sound and complete, see e.g. [Loz04,CGH05]. Such abstractions are similar to resource graphs from [GM05]. This is a variant of the automata-based approach introduced in [VW94] for plain LTL and further developed with concrete domains of interpretation in [DD07]. Surprisingly, the abstraction method used to establish these results does not scale to the whole logic, due to a subtle interplay between separation connectives and pointer arithmetic. Moreover, we provide new undecidability results for several problems, for instance $\text{SAT}^{ct}(\text{LF})$ (satisfiability with constant heap on the list fragment).

Related work. Previous temporal logics designed for pointer verification include Evolution Temporal Logic [YRSW03], based on the three-valued logic abstrac-

tion method that made the success of TVLA [LAS00], and Navigation temporal logic [DKR04], based on a tableau method quite similar to our automaton-based reduction. In these works, the assertion language for states is quite rich, as it includes for instance list predicate, quantification over addresses, and a freshness predicate. Because of this high expressive power, only incomplete abstractions are proposed, whereas we stick to exact methods. More importantly, our work addresses models with constant heaps and pointer arithmetic, which has not been done so far, and leads to a quite different perspective.

Omitted proofs can be found in the report [BDL07].

2 Memory Model and Specification Language

In this section, we introduce a separation logic dealing with pointer arithmetic and record values, and a temporal logic LTL^{mem} . Unlike BI's pointer logic from [IO01], we allow pointer arithmetic. Model-checking programs with pointer variables over LTL^{mem} specifications is our main problem of interest.

2.1 A separation logic with pointer arithmetic

Memory states. Let us introduce our model of memory. It captures features of programs with pointer variables that use pointer arithmetic and records. We assume a countably infinite set Var of variables (as usual, for a fixed formula we need only a finite amount), and an infinite set Val of values containing the set \mathbb{N} of naturals, thought as address indexes, and a special value nil . For simplicity, we assume that $\text{Val} = \mathbb{N} \uplus \{nil\}$. In order to model field selectors, we consider some infinite set Lab of labels. We will usually range over values with u, v , over naturals with i, j , over labels with $l, r, next, prev$, and over variables with x, y . In the remainder, we will assume some fixed injection $(x, i) \in \text{Var} \times \mathbb{N} \mapsto \langle x, i \rangle \in \text{Var}$.

We use the notation $E \rightarrow_{fin} F$ for the set of partial functions from E to F of finite domain; and $E \rightarrow_{fin+} F$ for the set of partial functions from E to F of finite and nonempty domain. The sets \mathcal{S} of stores and \mathcal{H} of heaps are then defined as follows: $\mathcal{S} \stackrel{\text{def}}{=} \text{Var} \rightarrow \text{Val}$ and $\mathcal{H} \stackrel{\text{def}}{=} \mathbb{N} \rightarrow_{fin} (\text{Lab} \rightarrow_{fin+} \text{Val})$. We will range over a store with s, s' and over a heap with h, h', h_1, h_2 . We call *memory state* a couple $(s, h) \in \mathcal{S} \times \mathcal{H}$.

We will refer to the domain of a heap h by $\text{dom}(h) \subseteq \mathbb{N}$. Intuitively, in our memory model, each index is thought as an entry point on some record cell containing several fields. Cells are either not allocated, or allocated with some record stored in. In a memory state (s, h) , the memory cell at index i is *allocated* if $i \in \text{dom}(h)$; in this case the stored record is $h(i) = \{l_1 \mapsto v_1, \dots, l_n \mapsto v_n\}$.

Note that the size of the information held in a memory cell is not fixed, nor bounded. Our models could be more concrete considering labels as offsets and relying on pointer arithmetic. But for our purpose, it will be convenient to consider pointer arithmetic independently.

Expressions	State Formulae
$e ::= x \mid \text{null}$	$\mathcal{A} ::= \pi$
Atomic formulae	$\mid \mathcal{A} * \mathcal{B} \mid \mathcal{A} \text{-} * \mathcal{B} \mid \text{emp}$ (spatial fragment)
$\pi ::= e = e' \mid e + i \xrightarrow{l} e$	$\mid \mathcal{A} \wedge \mathcal{B} \mid \mathcal{A} \rightarrow \mathcal{A} \mid \top \mid \perp$ (classical fragment)
Satisfaction	
$(s, h) \models_{\text{SL}} e = e'$	iff $\llbracket e \rrbracket_s = \llbracket e' \rrbracket_s$, with $\llbracket x \rrbracket_s = s(x)$ and $\llbracket \text{null} \rrbracket_s = \text{nil}$
$(s, h) \models_{\text{SL}} e + i \xrightarrow{l} e$	iff $\llbracket e \rrbracket_s \in \mathbb{N}$ and $\llbracket e \rrbracket + i \in \text{dom}(h)$ and $h(s(x) + i)(l) = \llbracket e \rrbracket_s$
$(s, h) \models_{\text{SL}} \text{emp}$	iff $\text{dom}(h) = \emptyset$
$(s, h) \models_{\text{SL}} \mathcal{A}_1 * \mathcal{A}_2$	iff $\exists h_1, h_2$ s.t. $h = h_1 * h_2$, $(s, h_1) \models_{\text{SL}} \mathcal{A}_1$ and $(s, h_2) \models_{\text{SL}} \mathcal{A}_2$
$(s, h) \models_{\text{SL}} \mathcal{A}' \text{-} * \mathcal{A}$	iff for all h' , if $h \perp h'$ and $(s, h') \models_{\text{SL}} \mathcal{A}'$ then $(s, h * h') \models_{\text{SL}} \mathcal{A}$
$(s, h) \models_{\text{SL}} \mathcal{A}_1 \wedge \mathcal{A}_2$	iff $(s, h) \models_{\text{SL}} \mathcal{A}_1$ and $(s, h) \models_{\text{SL}} \mathcal{A}_2$
$(s, h) \models_{\text{SL}} \mathcal{A}' \rightarrow \mathcal{A}$	iff $(s, h) \models_{\text{SL}} \mathcal{A}'$ implies $(s, h) \models_{\text{SL}} \mathcal{A}$
$(s, h) \models_{\text{SL}} \perp$	never and $(s, h) \models_{\text{SL}} \top$ always

Table 1. The syntax and semantics of SL with pointer arithmetic and records

Separation Logic. We now introduce the separation logic (SL) on top of which we will define our temporal logic. The syntax of the logic is given in Table 1.

In short, Separation logic is about reasoning on disjoint heaps, and we need to define what we mean by “disjoint heaps” in our model. We choose to allow to reason at the granularity of record cells, so that a record cell cannot be decomposed in disjoint parts. Let h_1 and h_2 be two heaps; we say that h_1 and h_2 are disjoint, noted $h_1 \perp h_2$, if $\text{dom}(h_1) \cap \text{dom}(h_2) = \emptyset$. The operation $h_1 * h_2$ is defined for disjoint heaps as the *disjoint union* of the two partial functions. Semantics of formulae is defined by the satisfaction relation \models_{SL} (see Table 1).

Formulae \mathcal{A} of SL are called *state formulae*. The size of the state formula \mathcal{A} , written $|\mathcal{A}|$, is the length of the string \mathcal{A} for some reasonably succinct encoding of variables and integers (binary representation). We will use the map $|\cdot|$ for other syntactic objects such as LTL^{mem} formulae. A formula $\mathcal{A} * \mathcal{B}$ with the *separation conjunction* states that \mathcal{A} holds on some portion of the memory heap and \mathcal{B} holds on a disjoint portion. A formula $\mathcal{A} \text{-} * \mathcal{B}$ states that the current heap, when extended with any disjoint heap verifying \mathcal{A} , will verify \mathcal{B} . Boolean operators are understood as usual. In the remainder, we focus on several specific fragments of this separation logic. We say that a formula is in the *record fragment* (RF) if all subformulae $x + i \xrightarrow{l} e$ use $i = 0$. In that case, we write $x \xrightarrow{l} e$. We say that a formula is in the *classical fragment* (CL) if it does not use the connectives $*$, $\text{-}*$. Finally, we say that a formula is in the *list fragment* (LF) if it is in the classical fragment and all subformulae $x + i \xrightarrow{l} e$ use $i = 0$ and $l = \text{next}$, and we may simply write $x \xrightarrow{l} e$. Clearly, the classical and record fragments are incomparable, while the list fragment is included in both of them.

Let us illustrate the expressive power of SL on examples. The formula $\text{-emp} * \text{-emp}$ means that at least two memory cells are allocated. The formula $x \xrightarrow{l} e$,

defined as $\neg(\neg\text{emp} * \neg\text{emp}) \wedge \mathbf{x} \xrightarrow{l} e$, is the local version of $\mathbf{x} \xrightarrow{l} e: s, h \models_{\text{SL}} \mathbf{x} \xrightarrow{l} e$ iff $\text{dom}(h) = \{s(\mathbf{x})\}$ and $h(s(\mathbf{x}))(l) = \llbracket e \rrbracket_s$. The formula $(\mathbf{x} \xrightarrow{l} \text{null}) * \perp$ is satisfied at (s_0, h_0) whenever there is no heap h_1 with $h_1 \perp h_0$ that allocates the variable \mathbf{x} to nil on l field, that is \mathbf{x} is allocated in h_0 .

\mathcal{A} is valid iff for every memory state (s, h) , we have $(s, h) \models_{\text{SL}} \mathcal{A}$ (written $\models_{\text{SL}} \mathcal{A}$). Satisfiability is defined dually.

Proposition 1. *The model-checking, satisfiability and validity problems for SL are PSPACE-complete.*

PSPACE-hardness results are consequences of [CYO01, Sect. 5.2]. The PSPACE upper bound for model-checking for SL is obtained by reduction to model-checking for RF that is shown in PSPACE thanks to forthcoming Lemma 2. Satisfiability for SL is reduced to model-checking for SL thanks to a small memory state property: every satisfiable state formulae \mathcal{A} can be satisfied by a memory state that can be encoded in polynomial size in \mathcal{A} .

2.2 Temporal extension

Memory states sequences Models of the logic LTL^{mem} are ω -sequences of memory states, that is elements in $(\mathcal{S} \times \mathcal{H})^\omega$ and they are understood as infinite computations of programs with pointer variables. In order to analyze computations from programs without destructive update, we shall also consider models with constant heap, that is elements in $\mathcal{S}^\omega \times \mathcal{H}$.

The logic LTL^{mem} . Formulae of LTL^{mem} are defined in Table 2. Atomic formulae of LTL^{mem} are state formulae from SL except that variables can be prefixed by the symbol “X”. For instance, Xx is interpreted by the value of \mathbf{x} at the

next memory state. We use the notation $\text{X}^i \mathbf{x}$ for $\overbrace{\text{X} \dots \text{X}}^{i \text{ times}} \mathbf{x}$ (but keep in mind that encoding $\text{X}^i \mathbf{x}$ requires memory space in $\mathcal{O}(i)$). The temporal operators are the standard next-time operator X and until operator U present in LTL, see e.g. [SC85]. The satisfaction relation $\rho, t \models \phi$ where ρ is a model of LTL^{mem} , $t \in \mathbb{N}$ and ϕ is a formula is also defined in Table 2. We use standard abbreviations such as $\text{F}\phi$, $\text{G}\phi \dots$. We freely use propositional variables p, q , having in mind that the propositional variable p should be understood as $\mathbf{x}_p = \mathbf{x}_\top$ for some fixed extra variables $\mathbf{x}_p, \mathbf{x}_q, \dots, \mathbf{x}_\top$.

Given a fragment Frag of SL, $\text{LTL}^{\text{mem}}(\text{Frag})$ is the restriction of LTL^{mem} to formulae in which occur only state formulae built over Frag (with extended variables $\text{X}^i \mathbf{x}$), and we write $\text{SAT}(\text{Frag})$ to denote the satisfiability problem for $\text{LTL}^{\text{mem}}(\text{Frag})$: given a temporal formula ϕ in $\text{LTL}^{\text{mem}}(\text{Frag})$, is there a model ρ such that $\rho, 0 \models \phi$? The variant problem in which we require that the model has a constant heap [resp. that the initial memory state is fixed, say (s, h)] is denoted by $\text{SAT}^{\text{ct}}(\text{Frag})$ [resp. $\text{SAT}_{\text{init}}(\text{Frag})$]. The problem $\text{SAT}_{\text{init}}^{\text{ct}}(\text{Frag})$ is defined analogously.

Enriched expressions	$\eta ::= \mathbf{x} \mid \mathbf{X}\eta \mid \mathbf{null}$
Atomic formulae	$\pi ::= \eta = \eta' \mid \eta + i \xrightarrow{l} \eta'$
State formulae	$\mathcal{A} ::= \pi \mid \mathbf{emp} \mid \mathcal{A} * \mathcal{B} \mid \mathcal{A} \text{-} * \mathcal{B} \mid \mathcal{A} \wedge \mathcal{B} \mid \mathcal{A} \rightarrow \mathcal{B} \mid \perp$
Temporal formulae	$\phi ::= \mathcal{A} \mid \mathbf{X}\phi \mid \phi \mathbf{U}\phi' \mid \phi \wedge \phi' \mid \neg\phi$
Semantics	
$\rho, t \models \mathbf{X}\phi$	iff $\rho, t + 1 \models \phi$.
$\rho, t \models \phi \mathbf{U}\phi'$	iff there is $t_1 \geq t$ s.t. $\rho, t_1 \models \phi'$ and $\rho, t' \models \phi$ for all $t' \in \{t, \dots, t_1 - 1\}$.
$\rho, t \models \phi \wedge \psi$	iff $\rho, t \models \phi$ and $\rho, t \models \psi$.
$\rho, t \models \neg\phi$	iff $\rho, t \not\models \phi$.
$\rho, t \models \mathcal{A}$	iff $s'_t, h_t \models_{\text{SL}} \mathcal{A}[\mathbf{X}^k \mathbf{x} \leftarrow (\mathbf{x}, k)]$ where $\rho = (s_t, h_t)_{t \geq 0}$ and s'_t is defined by $s'_t((\mathbf{x}, k)) = s_{t+k}(\mathbf{x})$.

Table 2. The syntax and semantics of LTL^{mem}

2.3 Programs with pointer variables

In this section, we define the model-checking problems for programs with pointer variables over LTL^{mem} specifications. The set \mathbf{I} of *instructions* used in the programs is defined by the grammar below:

$$\begin{aligned} \mathbf{instr} ::= & \mathbf{x} := \mathbf{y} \mid \mathbf{skip} \\ & \mid \mathbf{x} := \mathbf{y} \rightarrow l \mid \mathbf{x} \rightarrow l := \mathbf{y} \mid \mathbf{x} := \mathbf{cons}(l_1 : x_1, \dots, l_k : x_k) \mid \mathbf{free} \mathbf{x} \\ & \mid \mathbf{x} := \mathbf{y}[i] \mid \mathbf{x}[i] := \mathbf{y} \mid \mathbf{x} = \mathbf{malloc}(i) \mid \mathbf{free} \mathbf{x}, i \end{aligned}$$

The denotational semantics of an instruction \mathbf{instr} is defined as a partial function $\llbracket \mathbf{instr} \rrbracket : \mathcal{S} \times \mathcal{H} \rightarrow \mathcal{S} \times \mathcal{H}$, undefined when the instruction would cause a memory violation. We list in Table 3 the formal denotational semantics of our instruction set. Boolean combinations of equalities between expressions are called guards and its set is denoted by G . A program is defined as a triple (Q, δ, q_I) such that Q is a finite set of control states, q_I is the initial state and δ is the transition relation, a subset of $Q \times G \times \mathbf{I} \times Q$. We use $q \xrightarrow{g, \mathbf{instr}} q'$ to denote a transition. We say that a program is *without destructive update* if transitions are labeled only with instructions of the form $\mathbf{x} := \mathbf{y}$, $\mathbf{x} := \mathbf{y} \rightarrow l$, and $\mathbf{x} := \mathbf{y}[i]$. We write \mathbf{P} to denote the set of programs and \mathbf{P}^{ct} to denote the set of programs without destructive update.

A program is a finite object whose interpretation can be viewed as an infinite-state system. More precisely, given a program $\mathbf{p} = (Q, \delta, q_I)$, the transition system $\mathcal{S}_{\mathbf{p}} = (S, \rightarrow)$ is defined as follows: $S = Q \times (\mathcal{S} \times \mathcal{H})$ (set of configurations) and $(q, (s, h)) \rightarrow (q', (s', h'))$ iff there is a transition $q \xrightarrow{g, \mathbf{instr}} q' \in \delta$ such that $(s, h) \models g$ and $(s', h') = \llbracket \mathbf{instr} \rrbracket(s, h)$. Note that $\mathcal{S}_{\mathbf{p}}$ is not necessarily linear. A computation (or execution) of \mathbf{p} is defined as an infinite path in $\mathcal{S}_{\mathbf{p}}$ starting with control state q_I . Computations of \mathbf{p} can be viewed as LTL^{mem} models, using

$\llbracket \mathbf{x} := \mathbf{y} \rrbracket (s, h)$	$\stackrel{\text{def}}{=} (s[\mathbf{x} \mapsto s(\mathbf{y})], h).$
$\llbracket \mathbf{x} := \mathbf{y} \rightarrow l \rrbracket (s, h * \{i \mapsto \{l \mapsto v, \dots\}\})$	$\stackrel{\text{def}}{=} (s[\mathbf{x} \mapsto v], h * \{i \mapsto \{l \mapsto v, \dots\}\})$ with $s(\mathbf{y}) = i$
$\llbracket \mathbf{x} \rightarrow l := \mathbf{y} \rrbracket (s, h * \{i \mapsto \{l \mapsto v, \dots\}\})$	$\stackrel{\text{def}}{=} (s, h * \{i \mapsto \{l \mapsto s(\mathbf{y}), \dots\}\})$ with $s(\mathbf{x}) = i$
$\llbracket \mathbf{x} := \text{cons}(l_1 : \mathbf{x}_1, \dots, l_k : \mathbf{x}_k) \rrbracket (s, h)$	$\stackrel{\text{def}}{=} (s[\mathbf{x} \mapsto i], h * \{i \mapsto \{l_1 \mapsto s(\mathbf{x}_1), \dots, l_k \mapsto s(\mathbf{x}_k)\}\})$ with $i \notin \text{dom}(h)$
$\llbracket \text{free } \mathbf{x}, l \rrbracket (s, h * \{i \mapsto \{l \mapsto v, \dots\}\})$	$\stackrel{\text{def}}{=} (s, h * \{i \mapsto \{\dots\}\})$ with $s(\mathbf{x}) = i$
$\llbracket \text{skip} \rrbracket (s, h)$	$\stackrel{\text{def}}{=} (s, h)$
$\llbracket \mathbf{x} := \mathbf{y}[i] \rrbracket (s, h * \{i + i' \mapsto \{next \mapsto v\}\})$	$\stackrel{\text{def}}{=} (s[\mathbf{x} \mapsto v], h * \{i \mapsto \{next \mapsto v\}\})$ with $s(\mathbf{y}) = i'$
$\llbracket \mathbf{x}[i] := \mathbf{y} \rrbracket (s, h * \{i' + i \mapsto \{next \mapsto v\}\})$	$\stackrel{\text{def}}{=} (s, h * \{i + i' \mapsto \{next \mapsto s(\mathbf{y})\}\})$ with $s(\mathbf{x}) = i'$
$\llbracket \mathbf{x} := \text{malloc}(i) \rrbracket (s, h)$	$\stackrel{\text{def}}{=} (s[\mathbf{x} \mapsto i'], h * \{i' \mapsto \{next \mapsto nil\}\})$ $\dots * \{i' + i \mapsto \{next \mapsto nil\}\}$ with $i', \dots, i' + i \notin \text{dom}(h)$
$\llbracket \text{free } \mathbf{x}, i \rrbracket (s, h * \{i' + i \mapsto f\})$	$\stackrel{\text{def}}{=} (s, h) \text{ with } s(\mathbf{x}) = i'$

Table 3. Semantics for instructions

propositional variables to encode the extra information about the control states (details are omitted herein).

Model-checking aims at checking properties expressible in LTL^{mem} along computations of programs. To a logical fragment (SL, CL, RF, or LF), we associate a set of programs: all programs for SL and CL, programs with instructions having $i = 0$ for RF, and moreover with only the label *next* for LF. Given one of these fragments Frag of SL, we write $\text{MC}(\text{Frag})$ to denote the model-checking problem for Frag: given a temporal formula ϕ in LTL^{mem} with state formulae built over Frag and a program p of the associated fragment, is there an infinite computation ρ of p such that $\rho, 0 \models \phi$ (which we write $p \models \phi$)? The variant problem in which we require that the program is without destructive update [resp. that the initial memory state is fixed, say (s, h)] is denoted by $\text{MC}^{\text{ct}}(\text{Frag})$ [resp. $\text{MC}_{\text{init}}(\text{Frag})$]. The problem $\text{MC}_{\text{init}}^{\text{ct}}(\text{Frag})$ is defined analogously. We may write $p, (s, h) \models \phi$ to emphasize what is the initial memory state.

All the model-checking and satisfiability problems defined above can be placed in Σ_1^1 in the analytical hierarchy. Additionally, all the above problems can easily be shown PSPACE-hard since they all generalize LTL satisfiability and model-checking [SC85].

Using extended variables \mathbf{Xx} , we may express some programs as formulae. This actually holds only for programs without update, for the semantics with

constant heap. Intuitively, we express the control of the program with propositional variables, and define a formula that encode the transitions. To do so, we translate instructions of the form $x := y$ into $Xx = y$, $x := y \rightarrow l$ into $y \xrightarrow{l} Xx$, and $x := y[i]$ into $y + i \leftrightarrow Xx$. Guards are translated accordingly. As a consequence, the following result can be derived:

Lemma 1. *Let Frag be a fragment among SL, CL, RF, or LF. There is a logspace reduction from $MC^{ct}(\text{Frag})$ to $SAT^{ct}(\text{Frag})$ (resp. from $MC_{init}^{ct}(\text{Frag})$ to $SAT_{init}^{ct}(\text{Frag})$).*

3 Decidable Satisfiability Problems by Abstracting Computations

In this section we establish the PSPACE-completeness of the problems $SAT(\text{CL})$ and $SAT(\text{RF})$. To do so, we abstract memory states whose size is a priori unbounded by finite symbolic memory states. As usual, temporal infinity in models is handled by Büchi automata recognizing ω -sequences. We propose below an abstraction that is correct for CL (allowing pointer arithmetic) and for RF (allowing all operators from Separation Logic) taken separately but that is not exact for the full language SL.

3.1 Syntactic measures

The main approach to get decision procedures to verify infinite-state systems consists in introducing a symbolic representation for infinite sets of configurations. The symbolic representation defined below plays a similar role and has similarities with symbolic heaps for Separation Logic in Smallfoot [BCO05]. Let us start by some useful definitions. Following [Loz04], we introduce the set of *test formulae* that are formulae from SL of the forms below:

- $\text{alloc } x \stackrel{\text{def}}{=} (x \xrightarrow{\text{next}} \text{null}) \text{--} * \perp$ (x is allocated).
- $\text{size} \geq k \stackrel{\text{def}}{=} \overbrace{\neg \text{emp} * \dots * \neg \text{emp}}^{k \text{ times}}$ (at least k indices are allocated).
- $e + i \xrightarrow{l} e, e = e'$.

Given a formula ϕ of LTL^{mem} , we define its measure μ_ϕ understood as some pieces of information about the syntactic resources involved in ϕ . Indeed, forthcoming symbolic states are finite objects parameterized by such syntactic measures.

For a state formula \mathcal{A} of LTL^{mem} , the size of memory examined by \mathcal{A} , written $w_{\mathcal{A}}$, is inductively defined as follows: $w_{\mathcal{A}}$ is 1 for atomic formulae, $\max\{w_{\mathcal{A}_1}, w_{\mathcal{A}_2}\}$ for $\mathcal{A}_1 \wedge \mathcal{A}_2$ or $\mathcal{A}_1 \rightarrow \mathcal{A}_2$ or $\mathcal{A}_1 \text{--} * \mathcal{A}_2$, and $w_{\mathcal{A}_1} + w_{\mathcal{A}_2}$ for $\mathcal{A}_1 * \mathcal{A}_2$. Observe that $w_{\mathcal{A}} \leq |\mathcal{A}|$. Other simple sets about the syntactic resources of \mathcal{A} need to be defined: $\text{Lab}_{\mathcal{A}}$ is the set of labels from Lab occurring in \mathcal{A} , $\text{Var}_{\mathcal{A}}$

is the set of variables from Var occurring in \mathcal{A} , $\epsilon_{\mathcal{A}}$ is the set of natural numbers i such that $e + i \xrightarrow{l} e'$ occurs in \mathcal{A} and $m_{\mathcal{A}}$ is the maximal k such that $\mathbf{X}^k \mathbf{x}$ occurs in \mathcal{A} for some variable \mathbf{x} . A measure is defined as an element of $\mathbb{N} \times \mathcal{P}_f(\mathbb{N}) \times \mathbb{N} \times \mathcal{P}_f(\text{Lab}) \times \mathcal{P}_f(\text{Var})$ where $\mathcal{P}_f(X)$ denotes the set of finite subsets of some set X . The set of measures has a natural lattice structure for the pointwise order, noted below $\mu \leq \mu'$. We also write $\mu[w \leftarrow 0]$ to denote the measure μ except that $w = 0$.

The measure for \mathcal{A} , written $\mu_{\mathcal{A}}$, is the tuple $(m_{\mathcal{A}}, \epsilon_{\mathcal{A}}, w_{\mathcal{A}}, \text{Lab}_{\mathcal{A}}, \text{Var}_{\mathcal{A}})$. The measure of some formula ϕ of LTL^{mem} , written μ_{ϕ} , is $\sup\{\mu_{\mathcal{A}} : \mathcal{A} \text{ occurs in } \phi\}$.

Definition 1. Given a measure $\mu = (m, \epsilon, w, X, Y)$, we write \mathcal{T}_{μ} to denote the finite set of test formulae ψ of the grammar:

$$\begin{aligned} e ::= \langle \mathbf{x}, u \rangle \mid \text{null} \quad f ::= e + i \\ \psi ::= f \xrightarrow{l} e \mid \text{alloc } f \mid e = e' \mid \text{size} \geq k \end{aligned}$$

with $u \leq m$, $i \in \epsilon$, $l \in X$, $k < w$ and $\mathbf{x} \in Y$.

Observe that the cardinal of $\mathcal{T}_{\mu_{\phi}}$ is polynomial in $|\phi|$. Given a measure $\mu = (m, \epsilon, w, X, Y)$ and a memory state (s, h) , we write $\text{Abs}_{\mu}(s, h) = \{\mathcal{A} \in \mathcal{T}_{\mu} : (s, h) \models_{\text{SL}} \mathcal{A}\}$ to denote the abstraction of (s, h) wrt μ . Given a measure μ and two memory states (s, h) and (s', h') , we write $(s, h) \simeq_{\mu} (s', h')$ iff $\text{Abs}_{\mu}(s, h) = \text{Abs}_{\mu}(s', h')$, that is formulae in \mathcal{T}_{μ} cannot distinguish the two memory states. Lemma 2 below states that our abstraction is correct for CL and RF.

Lemma 2. Let (s, h) and (s', h') be two memory states such that $(s, h) \simeq_{\mu} (s', h')$ [resp. $(s, h) \simeq_{\mu[w \leftarrow 0]} (s', h')$]. For any state formula \mathcal{A} such that $\mu_{\mathcal{A}} \leq \mu$ and \mathcal{A} belongs to RF [resp. CL], we have $(s, h) \models_{\text{SL}} \mathcal{A}$ iff $(s', h') \models_{\text{SL}} \mathcal{A}$.

Note that we can extend this result to the whole SL by considering test formulae of the form $e + i = e' + j$.

3.2 Symbolic models

We write Σ_{μ} to denote the powerset of \mathcal{T}_{μ} ; Σ_{μ} is thought as an alphabet, and elements $a \in \Sigma_{\mu}$ are called *letters*. A symbolic model wrt μ is defined as an infinite sequence $\sigma \in \Sigma_{\mu}^{\omega}$. Symbolic models are abstractions of models from LTL^{mem} : given a model $\rho : \mathbb{N} \rightarrow \mathcal{S} \times \mathcal{H}$ and a measure μ , we write $\text{Abs}_{\mu}(\rho) : \mathbb{N} \rightarrow \Sigma_{\mu}$ to denote the symbolic model wrt μ such that for any t , $\text{Abs}_{\mu}(\rho)(t) \stackrel{\text{def}}{=} \{\mathcal{A} \in \mathcal{T}_{\mu} : \rho, t \models \mathcal{A}[\langle \mathbf{x}, u \rangle \leftarrow \mathbf{X}^u \mathbf{x}]\}$.

To a letter a , we associate the formula $\mathcal{A}_a = \bigwedge_{\mathcal{A} \in a} \mathcal{A} \wedge \bigwedge_{\mathcal{A} \notin a} \neg \mathcal{A}$. For σ a symbolic model, and ϕ a formula such that $\mu_{\phi} \leq \mu$, we define the symbolic satisfaction relation $\sigma, t \models_{\mu} \phi$ as satisfaction for models except for the clause about atomic subformulae that becomes: $\sigma, t \models_{\mu} \mathcal{A}$ iff $\models_{\text{SL}} \mathcal{A}_{\sigma(t)} \Rightarrow \mathcal{A}[\mathbf{X}^u \mathbf{x} \leftarrow \langle \mathbf{x}, u \rangle]$. We write $L^{\mu}(\phi)$ to denote the set of symbolic models σ wrt μ such that $\sigma, 0 \models_{\mu} \phi$. As a corollary of Lemma 2, we get a soundness result for our abstraction:

Proposition 2. *Let ϕ be a formula of $\text{LTL}^{\text{mem}}(\text{RF})$ [resp. of $\text{LTL}^{\text{mem}}(\text{CL})$] and $\mu_\phi \leq \mu$. For any model ρ , we have that $\rho \models \phi$ iff $\text{Abs}_\mu(\rho) \models \phi$ [resp. $\text{Abs}_{\mu[w \leftarrow 0]}(\rho) \models \phi$].*

Note that Abs_μ is not surjective; we note $\text{L}_{\text{sat}}^\mu$ the set of symbolic models wrt μ that are abstractions of some model of LTL^{mem} . Consequently, ϕ in $\text{LTL}^{\text{mem}}(\text{RF})$ is satisfiable iff $\text{L}^{\mu_\phi}(\phi) \cap \text{L}_{\text{sat}}^{\mu_\phi}$ is nonempty.

3.3 ω -regularity and PSPACE upper bound

In order to show that $\text{SAT}(\text{RF})$ and $\text{SAT}(\text{CL})$ are in PSPACE we shall explain why testing the nonemptiness of $\text{L}^{\mu_\phi}(\phi) \cap \text{L}_{\text{sat}}^{\mu_\phi}$ can be done in PSPACE. Below we treat explicitly the case for RF. For CL, replace every occurrence of μ_ϕ by $\mu_\phi[w \leftarrow 0]$. To do so, we show that each language can be recognized by an exponential-size Büchi automaton satisfying the good properties to establish the PSPACE upper bound. If \mathbb{A} is a Büchi automaton, we note $\text{L}(\mathbb{A})$ the language recognized by \mathbb{A} . Following [VW94,DD07], let \mathbb{A} be the generalized Büchi automaton defined by the structure $(\Sigma, Q, \delta, I, \mathcal{F})$ s.t.:

- Q is the set of so-called atoms of ϕ , that are sets of temporal formulae included in the so-called closure set $\text{cl}(\phi)$ (see [VW94]), $I = \{X \in Q : \phi \in X\}$, and $\Sigma = \Sigma_\mu$.
- $X \xrightarrow{a} Y$ iff
 1. for every atomic formula \mathcal{A} of X , $\models_{\text{SL}} \mathcal{A}_a \Rightarrow \mathcal{A}[X^u \mathbf{x} \leftarrow \langle \mathbf{x}, u \rangle]$.
 2. for every $X\phi' \in \text{cl}(\phi)$, $X\phi' \in X$ iff $\phi' \in Y$.
- Let $\{\phi_1 \cup \phi'_1, \dots, \phi_n \cup \phi'_n\}$ be the set of until formulae in $\text{cl}(\phi)$. We pose $\mathcal{F} = \{F_1, \dots, F_n\}$ where $F_i = \{X \in Q : \phi_i \cup \phi'_i \notin X \text{ or } \phi'_i \in X\}$ for $i \in \{1, \dots, n\}$.

Let \mathbb{A}_ϕ^μ be the Büchi automaton equivalent to the generalized Büchi automaton \mathbb{A} . It is easy to observe that \mathbb{A}_ϕ^μ has an exponential amount of states in the size of ϕ and its transition relation can be checked in polynomial space in the size of ϕ . Moreover,

Lemma 3. *Let ϕ in $\text{LTL}^{\text{mem}}(\text{RF})$ [resp. $\text{LTL}^{\text{mem}}(\text{CL})$] and $\mu \geq \mu_\phi$ [resp. $\mu \geq \mu_\phi[w \leftarrow 0]$]. Then, $\text{L}(\mathbb{A}_\phi^\mu) = \text{L}^\mu(\phi)$.*

We can also build a Büchi automaton $\mathbb{A}_{\text{sat}}^\mu$ such that $\text{L}(\mathbb{A}_{\text{sat}}^\mu) = \text{L}_{\text{sat}}^\mu$. $\mathbb{A}_{\text{sat}}^\mu$ is defined as $(\Sigma, Q, \delta, I, F)$, where $\Sigma = \Sigma_\mu$, $Q = \Sigma_\mu$, $F = I = Q$ and $a \xrightarrow{a'} a''$ iff:

1. $\mathcal{A}_a, \mathcal{A}_{a''}$ are satisfiable, and $a = a'$,
2. for every formula $\langle \mathbf{x}, u \rangle = \langle \mathbf{x}', u' \rangle \in \mathcal{T}_\mu$ with $u, u' \geq 1$, $\langle \mathbf{x}, u \rangle \in a$ iff $\langle \mathbf{x}, u - 1 \rangle = \langle \mathbf{x}', u' - 1 \rangle \in a''$.

If $\mu = \mu_\phi$, then $\mathbb{A}_{\text{sat}}^\mu$ is of exponential-size in $|\phi|$ and the transition relation can be checked in polynomial space in $|\phi|$. More importantly, this automaton recognizes satisfiable symbolic models.

Lemma 4. *Let ϕ in $\text{LTL}^{\text{mem}}(\text{RF})$ [resp. $\text{LTL}^{\text{mem}}(\text{CL})$] and $\mu = \mu_\phi$ [resp. $\mu = \mu_\phi[w \leftarrow 0]$]. Then, $\text{L}(\mathbb{A}_{\text{sat}}^\mu) = \text{L}_{\text{sat}}^\mu$.*

This lemma is essential and it is not possible to extend it to the whole logic LTL^{mem} even by allowing test formulae of the form $x + i = y + j$ since we would need automata with counters. Now, we can state our main complexity result.

Theorem 1. *SAT(RF) and SAT(CL) are PSPACE-complete.*

Proof. (sketch) The lower bound is from LTL [SC85]. Let ϕ be an instance formula of SAT(RF) (for SAT(CL) replace below μ_ϕ by $\mu_\phi[w \leftarrow 0]$). As seen earlier, ϕ is satisfiable iff $L^{\mu_\phi}(\phi) \cap L_{sat}^{\mu_\phi}$ is nonempty. Hence, ϕ is satisfiable iff $L(\mathbb{A}_\phi^{\mu_\phi}) \cap L(\mathbb{A}_{sat}^{\mu_\phi}) \neq \emptyset$. The intersection automaton is of exponential size in the size of ϕ and can be checked nonempty by a nondeterministic on-the-fly algorithm. Since nonemptiness problem for Büchi automata is NLOGSPACE-complete and the transition relation in the intersection automaton can be checked in polynomial space in $|\phi|$, we obtain a nondeterministic polynomial space algorithm for testing satisfiability of ϕ . By Savitch's theorem, we get the PSPACE upper bound. \square

3.4 Other problems in PSPACE

Let Frag be either the classical fragment or the record fragment. Lemma 1 provides a reduction from $MC_{init}^{ct}(\text{Frag})$ to $SAT_{init}^{ct}(\text{Frag})$ based on a program-as-formula encoding. As we will see now, we may also reduce $SAT_{init}^{ct}(\text{Frag})$ to $SAT(\text{Frag})$ internalizing an approximation of the initial memory state whose logical language cannot distinguish from the initial memory state. As a consequence, the PSPACE upper bound for $SAT(\text{Frag})$ entails the PSPACE upper bound for both $SAT_{init}^{ct}(\text{Frag})$ and $MC_{init}^{ct}(\text{Frag})$.

Proposition 3. *The problems $SAT_{init}^{ct}(\text{RF})$, $MC_{init}^{ct}(\text{RF})$, $SAT_{init}^{ct}(\text{CL})$ and $MC_{init}^{ct}(\text{CL})$ are PSPACE-complete.*

Proof. By Lemma 1 and since $SAT_{init}^{ct}(\text{RF})$ is known to be PSPACE-hard, it remains to establish the PSPACE upper bound for $SAT_{init}^{ct}(\text{RF})$.

Given a formula ϕ and an initial memory state (s, h) , we shall build in polynomial-time a formula $\phi_{s,h}^{ct}$ in SAT(RF) such that ϕ is satisfiable in a model with initial memory state (s, h) and constant heap iff $\phi_{s,h}^{ct}$ is satisfiable by a general model. Since we have shown that SAT(RF) is in PSPACE, this guarantees that $SAT_{init}^{ct}(\text{RF})$ is in PSPACE. The idea of the proof is to internalize the initial memory state and the fact that the heap is constant in the logic SAT(RF). Actually, one cannot exactly express that the heap is constant (see details below) but the approximation we use will be sufficient for our purpose.

Apart from the variables of ϕ , the formula $\phi_{s,h}^{ct}$ is built over additional variables in $V = \{x_i : i \in \text{dom}(h) \cup \text{Im}(s)\} \cup \{x_{i,l} : i \in \text{dom}(h), l \in \text{dom}(h(i))\}$. The formula $\phi_{s,h}^{ct}$ is of the form $G(\psi_1 \wedge \psi_2 \wedge \psi_3) \wedge \psi_s \wedge \psi'$, where the subformulae are defined as follows.

- ψ_1 states that the heap is almost equal to h since we cannot forbid in the logical language additional labels ($\text{dom}(h) = \{i_1, \dots, i_k\}$):

$$\psi_1 \stackrel{\text{def}}{=} \left(\bigwedge_{l \in \text{dom}(h(i_1))} x_{i_1} \stackrel{l}{\mapsto} x_{i_1,l} \right) * \dots * \left(\bigwedge_{l \in \text{dom}(h(i_k))} x_{i_k} \stackrel{l}{\mapsto} x_{i_k,l} \right).$$

- ψ_2 states which variables are equal and which ones are not, depending on the initial memory state. By way of example, for $i \neq j \in \text{dom}(h)$, a conjunct of ψ_2 is $x_i \neq x_j$. Similarly, if $h(i)(l) = j$ and $j \in \text{dom}(h)$ then $x_{i,l} = x_j$ is a conjunct of ψ_2 . Details are omitted.
- ψ_3 states that the auxiliary variables remain constant: $\bigwedge_{x \in V} x = Xx$.
- The formula ψ' is obtained from ϕ by replacing each occurrence of $x \xrightarrow{l} e$ by

$$x \xrightarrow{l} e \wedge \bigwedge_{i \in \text{dom}(h), l \notin \text{dom}(h(i))} x \neq x_i.$$

The additional conjunct is useful because our logical language cannot state that a label is not in the domain of some allocated address.

- ψ_s states constraints about the initial store s : $\psi_s \stackrel{\text{def}}{=} \bigwedge_{x \in \phi} x = x_{s(x)}$.

It is then easy to check that ϕ is satisfiable in a model with initial memory state (s, h) with constant heap iff $\phi_{s,h}^{ct}$ is satisfiable by a general model.

As far as the results for the classical fragment are concerned, by Lemma 1, there is a logspace reduction from $\text{MC}_{init}^{ct}(\text{CL})$ to $\text{SAT}_{init}^{ct}(\text{CL})$ and as done above one can reduce $\text{SAT}_{init}^{ct}(\text{CL})$ to $\text{SAT}(\text{CL})$. \square

4 Undecidability Results

In this section, we show several undecidability results by using reduction from problems for Minsky machines. So, first, we recall that a Minsky machine M consists of two counters C_1 and C_2 , and a sequence of $n \geq 1$ instructions, each of which may increment or decrement one of the counters, or jump conditionally upon the counters being zero. The l^{th} instruction (l is its location counter) has one of the following forms either “ $l: C_i := C_i + 1$; goto l' ” or “ $l: \text{if } C_i = 0 \text{ then goto } l' \text{ else } C_i := C_i - 1$; goto l'' ”. In a nondeterministic machine, after an incrementation or a decrementation a nondeterministic choice of the form “goto l_1 or goto l_2 ” is performed.

The configurations of M are triples (l, c_1, c_2) , where $1 \leq l \leq n$, $c_1 \geq 0$, and $c_2 \geq 0$ are the current values of the location counter and the two counters C_1 and C_2 , respectively. The consecution relation on configurations is defined in the obvious way. A computation of M is a sequence of related configurations, starting with the initial configuration $(1, 0, 0)$.

Different encodings of counters are used here. For instance, in [BFLS06], a counter C with value n is represented by a list of length n pointed to by a x dedicated to C . The same idea is used in the proof of Proposition 4 below. In order to show undecidability of $\text{SAT}(\text{SL})$ we alternatively encode counters by relying on pointer arithmetic and properties of heaps. Programs without destructive updates can simulate finite computations of Minsky machines by guessing at the start of the computation the maximal value of counters (encoded by a list of the length of the maximal value). As a consequence,

Proposition 4. $\text{SAT}^{ct}(\text{LF})$ and $\text{MC}^{ct}(\text{LF})$ are Σ_1^0 -complete.

By contrast, programs with destructive update can work with unbounded heaps, and by using the representation of counters as above, they can faithfully simulate a Minsky machine even if an empty heap is the initial heap. Because LTL can express repeated accessibility, Σ_1^1 -hardness can be obtained.

Proposition 5. *The problems $\text{MC}(\text{LF})$ and $\text{MC}_{init}(\text{LF})$ are Σ_1^1 -complete.*

Let us briefly explain how to encode incrementation and decrementation with separating connectives and pointer arithmetic. Observe that expressions of the form $x = y + 1$ are not allowed in the logical language. We solve this point in two different ways: using non-aliasing expressed by the separating conjunction, and using the precise pointing assertion $x \xrightarrow{\text{next}} \eta$ stating that the heap contains only one cell, in conjunction with the $*$ operator.

$$\begin{aligned}\phi_{x++}^* &= (\text{Xx} \xrightarrow{\text{next}} \text{null} \wedge x + 1 \xrightarrow{\text{next}} \text{null}) \wedge \neg(\text{Xx} \xrightarrow{\text{next}} \text{null} * x + 1 \xrightarrow{\text{next}} \text{null}) \\ \phi_{x--}^* &= (\text{Xx} + 1 \xrightarrow{\text{next}} \text{null} \wedge x \xrightarrow{\text{next}} \text{null}) \wedge \neg(\text{Xx} + 1 \xrightarrow{\text{next}} \text{null} * x \xrightarrow{\text{next}} \text{null}) \\ \phi_{x++}^{-*} &= \text{emp} \wedge ((\text{Xx} \xrightarrow{\text{next}} \text{null}) * x + 1 \xrightarrow{\text{next}} \text{null}) \\ \phi_{x--}^{-*} &= \text{emp} \wedge ((x \xrightarrow{\text{next}} \text{null}) * \text{Xx} + 1 \xrightarrow{\text{next}} \text{null})\end{aligned}$$

The formulae based on the separating conjunction correctly express incrementation and decrementation when the cells at index $x, x + 1, x - 1$ are allocated, whereas formulae based on the operator $*$ do not need the same assumption.

Let $\text{SAT}_?^?(SL)$ be any satisfiability problem among the four variants.

Proposition 6. $\text{SAT}_?^?(SL)$ is Σ_1^1 -complete.

Proof. We reduce the recurrence problem for nondeterministic Minsky machines [AH94] to $\text{SAT}_?^?(SL)$. Let ϕ_0 be the formula $\text{G}(\text{emp} \wedge \bigwedge_{i=1}^2 (x_i \neq \text{null}))$. Incrementation and decrementation are performed thanks to ϕ_{x++}^{-*} and ϕ_{x--}^{-*} , respectively. For any model ρ such that $\rho, 0 \models \phi_0$, and for any t , we have $\rho, t \models \phi_{x_i++}^{-*}$ iff $s_t(x_i) + 1 = s_{t+1}(x_i)$. Hence, we have a means to encode incrementation. Similarly, $\rho, t \models \phi_{x_i--}^{-*}$ and $s_t(x_i) > 0$ iff $s_t(x_i) - 1 = s_{t+1}(x_i)$. The fact that a counter does not change is encoded by $x_i = \text{Xx}_i$. Given that $\phi_1 = \text{G}(x_{zero} = \text{Xx}_{zero} \wedge x_{zero} \neq \text{null})$ holds, zero tests are encoded by $x_i = x_{zero}$.

Given a nondeterministic Minsky machine M , we write ψ_l to denote the formula encoding instruction l . For instance for the instruction “ l : if $C_1 = 0$ then goto l' else $C_1 := C_1 - 1$; goto l'_1 or goto l'_2 ” ψ_l is equal to $\text{G}((l \wedge x_1 \neq x_{zero}) \Rightarrow (x_2 = \text{Xx}_2 \wedge (\text{Xl}'_1 \vee \text{Xl}'_2) \wedge \phi_{x_1--}^{-*})) \wedge \text{G}((l \wedge x_1 = x_{zero}) \Rightarrow (x_1 = \text{Xx}_1 \wedge x_2 = \text{Xx}_2 \wedge \text{Xl}'))$. Hence, $(x_1 = x_2 = x_{zero}) \wedge \phi_0 \wedge \phi_1 \wedge \bigwedge_l \psi_l \wedge \text{GF}n$ is satisfiable iff M has a computation with location counter n repeated infinitely often. \square

Proposition 7. *The problem $\text{SAT}(SL \setminus \{-*\})$ is Σ_1^1 -complete.*

The proof of Proposition 7 is similar to the proof of Theorem 6 except that incrementation and decrementation are performed with the formulae ϕ_{x++}^* and ϕ_{x--}^* , respectively.

5 Conclusion

In the paper, we have introduced a temporal logic LTL^{mem} for which assertion language is quantifier-free separation logic. Figure 1 contains a summary of the complexity results about satisfiability and model-checking problems for the fragments LF, CL and RF. Σ_1^1 -completeness results for $SAT_{\exists}^1(\text{SL})$, $SAT(\text{SL} \setminus \{\rightarrow\})$ and $\text{MC}(\text{LF})$ can be found in Propositions 6, 7, and 5, respectively. A thin and straight [resp. bold and curved] arrow between a source problem and a target problem means that the upper [resp. lower] bound for the target problem is shown thanks to the upper [resp. lower] bound for the source problem.

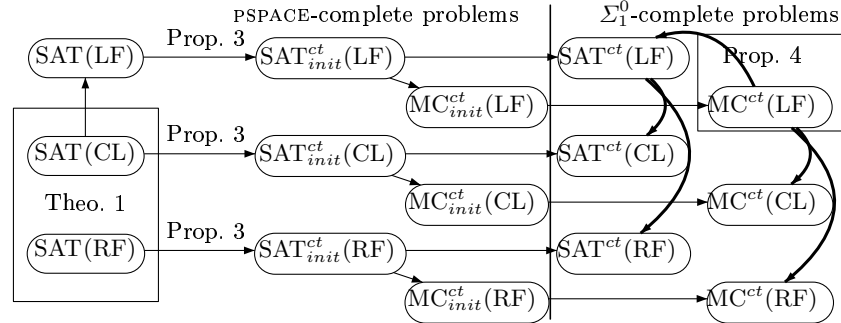


Fig. 1. Complexity of reasoning tasks with LTL^{mem}

Finally, extending LTL^{mem} with a special propositional variable $\text{heap}^=$ stating that the current heap is equal to the next one, can lead to undecidability (look at the problems of the form $SAT_{\exists}^1(\text{Frag})$). However, it is open whether satisfiability becomes decidable if we restrict the interplay between the “until” operator U and $\text{heap}^=$, for instance to forbid subformulae of the form $G \text{heap}^=$ with positive polarity.

References

- [AH94] R. Alur and T.A. Henzinger. A really temporal logic. *JACM*, 41:181–204, 1994.
- [BBH⁺06] A. Bouajjani, M. Bozga, P. Habermehl, R. Iosif, P. Moro, and T. Vojnar. Programs with lists are counter automata. In *CAV’06*, volume 4144 of *LNCS*, pages 517–531. Springer, 2006.
- [BCMS01] O. Burkart, D. Caucal, F. Moller, and B. Steffen. Verification of infinite structures. In *Handbook of Process Algebra*, pages 545–623. Elsevier, 2001.
- [BCO05] J. Berdine, C. Calcagno, and P. W. O’Hearn. Symbolic execution with separation logic. *APLAS’05*, 3780:52–68, 2005.
- [BDL07] R. Brochenin, S. Demri, and E. Lozes. Reasoning about sequences of memory states. Technical report, LSV, ENS de Cachan, 2007.

- [BFLS06] S. Bardin, A. Finkel, E. Lozes, and A. Sangnier. From pointer systems to counter systems using shape analysis. *5th International Workshop on Automated Verification of Infinite-State Systems (AVIS'06)*, 2006.
- [BFN04] S. Bardin, A. Finkel, and D. Nowak. Toward symbolic verification of programs handling pointers. In *3rd International Workshop on Automated Verification of Infinite-State Systems (AVIS'04)*, 2004.
- [BIL04] M. Bozga, R. Iosif, and Y. Lakhnech. On logics of aliasing. In *SAS'04*, volume 3148 of *LNCS*, pages 344–360. Springer, 2004.
- [CC00] H. Comon and V. Cortier. Flatness is not a weakness. *CSL'00*, 1862:262–276, 2000.
- [CGH05] C. Calcagno, Ph. Gardner, and M. Hague. From separation logic to first-order logic. In *FOSSACS'05*, volume 3441 of *LNCS*, pages 395–409. Springer, 2005.
- [CYO01] C. Calcagno, H. Yang, and P. O'Hearn. Computability and complexity results for a spatial assertion language for data structures. In *FST&TCS'01*, volume 2245 of *LNCS*, pages 108–119. Springer, 2001.
- [DD07] S. Demri and D. D'Souza. An automata-theoretic approach to constraint LTL. *Information and Computation*, 205(3):380–415, 2007.
- [DKR04] D. Distefano, J.-P. Katoen, and A. Rensink. Who is pointing when to whom? on the automated verification of linked list structures. In *FST&TCS'04*, volume 3328 of *LNCS*, pages 250–262. Springer, 2004.
- [GKWZ03] D. Gabbay, A. Kurucz, F. Wolter, and M. Zakharyashev. *Many-dimensional modal logics: theory and applications*. CUP, 2003.
- [GM05] D. Galmiche and D. Mery. Characterizing provability in BI's pointer logic through resource graphs. In *LPAR'05*, volume 3835 of *LNCS*, pages 459–473. Springer, 2005.
- [IO01] S. Ishtiaq and P. O'Hearn. BI as an assertion language for mutable data structures. In *POPL'01*, pages 14–26, 2001.
- [JJKS97] J. Jensen, M. Jorgensen, N. Klarlund, and M. Schwartzbach. Automatic verification of pointer programs using monadic second-order logic. In *PLDI'97*, pages 226–236. ACM, 1997.
- [LAS00] T. Lev-Ami and M. Sagiv. TVLA: A system for implementing static analyses. In *SAS'00*, pages 280–301, 2000.
- [Loz04] E. Lozes. Separation logic preserves the expressive power of classical logic. In *2nd Workshop on Semantics, Program Analysis, and Computing Environments for Memory Management (SPACE'04)*, 2004.
- [Pnu77] A. Pnueli. The temporal logic of programs. In *FOCS'77*, pages 46–57. IEEE, 1977.
- [Rey02] J.C. Reynolds. Separation logic: a logic for shared mutable data structures. In *LICS'02*, pages 55–74. IEEE, 2002.
- [SC85] A. Sistla and E. Clarke. The complexity of propositional linear temporal logic. *JACM*, 32(3):733–749, 1985.
- [VW94] M. Vardi and P. Wolper. Reasoning about infinite computations. *Information and Computation*, 115:1–37, 1994.
- [YRSW03] E. Yahav, Th. Reps, M. Sagiv, and R. Wilhelm. Verifying temporal heap properties specified via evolution logic. In *ESOP'03*, volume 2618 of *LNCS*, pages 204–22. Springer, 2003.