Practical Abstraction for Model Checking of Multi-Agent Systems

Wojciech Jamroga\textsuperscript{1,2}, Yan Kim\textsuperscript{1}

\textsuperscript{1}Interdisciplinary Centre for Security, Reliability, and Trust, SnT, University of Luxembourg
\textsuperscript{2}Institute of Computer Science, Polish Academy of Science, Warsaw, Poland
{wojciech.jamroga, yan.kim}@uni.lu

Abstract
Model checking of multi-agent systems (MAS) is known to be hard, both theoretically and in practice. A smart abstraction of the state space may significantly reduce the model, and facilitate the verification. We propose and study an intuitive agent-based abstraction scheme, based on the removal of variables in the representation of a MAS. This allows to achieve a desired reduction of a state space without generating the global model of the system. Moreover, the process is easy to understand and control even for domain experts with little knowledge of computer science. We formally prove the correctness of the approach, and evaluate the gains experimentally on a family of postal voting models.

1 Introduction
Multi-agent systems (MAS) describe interactions of autonomous agents, often assumed to be intelligent and/or rational. The theoretical foundations of MAS are mostly based on modal logic and game theory (Wooldridge 2002; Shoham and Leyton-Brown 2009). In particular, the temporal logics CTL, LTL, and CTL* provide formalizations of many relevant properties, including reachability, liveness, safety, and fairness (Emerson 1990). Algorithms and tools for verification have been in constant development for 40 years, with temporal model checking being the most popular approach (Baier and Katoen 2008; Clarke et al. 2018).

Complexity and state-space explosion. However, formal verification of MAS is known to be hard, both theoretically and in practice. The state-space explosion is a major obstacle here, as faithful models of real-world systems are huge and infeasible even to generate, let alone verify. In consequence, model checking of MAS with respect to their modular representations ranges from PSPACE-complete to undecidable (Schnoebelen 2003; Bulling et al. 2010). No less importantly, it is often unclear how to create the input model, especially if the system to be modelled involves human behaviour (Jamroga et al. 2020b). Specification is error-prone and difficult to debug and validate, and most model-checkers for MAS do not even have a graphical user interface.\textsuperscript{1} In realistic cases, one does not really know if what is verified and what we think we verify are indeed the same thing.

Dealing with state-space explosion. Much work has been done to contain the state-space explosion by smart representation and/or reduction of input models. Symbolic model checking based on SAT- or BDD-based representations of the state/transition space (McMillan 1993; McMillan 2002; Penczek and Lomuscio 2003; Kacprzak et al. 2004; Lomuscio and Penczek 2007; Huang and van der Meyden 2014; Lomuscio et al. 2017) fall into the former group. Model reduction methods include partial-order reduction (Peled 1993; Gerth et al. 1999; Jamroga et al. 2020a), equivalence-based reductions (de Bakker et al. 1984; Alur et al. 1998; Belardinelli et al. 2021), and state abstraction (Cousot and Cousot 1977), see Section 2 for a detailed discussion.

Towards practical abstraction. A smart abstraction of the state space may reduce the model to manageable size by clustering “similar” concrete states into abstract states, which should facilitate verification. Unfortunately, such clustering may remove essential information from the model, thus making the verification of the abstract model inconclusive for the original model. Lossless abstractions can be obtained by means of abstraction-refinement (Clarke et al. 2000) but, typically, they are difficult to compute or provide insufficient reduction of the model – quite often both.

In consequence, one has to live with abstractions that only approximate the concrete model. Moreover, crafting a good abstraction is an art that relies on the domain expertise of the modeller. Since domain experts are seldom computer scientists or specialists in formal methods, the theoretical formulation of abstraction as an arbitrary mapping from the concrete to the abstract state space has little appeal. Moreover, model checking tools typically do not support abstraction, so doing one would require to manipulate the input specification code, which is a difficult task in itself. What we need is a simple and intuitive methodology for selecting information to be removed from a MAS model, and for its automated removal that preserves certain guarantees. Last but not least, practical abstraction should be applied on modular representations of MAS, unlike the theoretical concept that is usually defined with respect to explicit models of global states.

Contribution. In this paper, we suggest that the conceptually simplest kind of abstraction consists in removing a domain

\textsuperscript{1}Notable exceptions include UPPAAL (Behrmann et al. 2004) and STV (Kurpiewski et al. 2021).
variable from the specification of the input model. This can be
generalized to the merging of several variables into a single
one, and possibly clustering their valuations. It is also natural
to restrict the scope of abstraction to a part of the input graph.
As the main technical contribution, we propose a correct-by-
design method to generate such abstractions. We prove that
the abstractions preserve the valuations of temporal formulae
in Universal $\text{CTL}^*$ ($\text{ACTL}^*$). More precisely, our $\text{may}$-abstractions preserve the falsity of $\text{ACTL}^*$ properties, so if
$\varphi \in \text{ACTL}^*$ holds in the abstract model, it must also hold in
the original one. Conversely, our $\text{must}$-abstractions preserve the
truth of $\text{ACTL}^*$ formulae, so if $\varphi \in \text{ACTL}^*$ is false
in the abstract model, it must also be false in the original one. We evaluate the efficiency of the method by verifying a
scalable model of postal voting in UPPAAL. The experiments
show that the method is user-friendly, compatible with a
state of the art verification tool, and capable of providing
significant computational gains.

2 Related Work

State abstraction was introduced in the 1970s (Couso and
Couso 1977), and studied intensively in the context of tempo-
ral properties (Clarke et al. 1994; Godefroid and Jagadeesan
2002). Automatically generated lossless abstractions have
been defined through abstraction-refinement (Dams and
Grumberg 2018; Clarke et al. 2000; Shoahm and Grumberg
2004). In particular, counterexample-guided abstraction re-
finement was proposed in (Clarke et al. 2000; Clarke et al.
2003), and implemented in NuSMV (Cimatti et al. 2002).
Unfortunately, lossless abstraction often results in abstract
models that are still too large for practical verification. In
this paper, we focus on lossy $\text{may}$/$\text{must}$ abstractions, based
on user-defined equivalence relations.

This kind of abstractions have been studied in (Dams et al.
1997; Godefroid et al. 2001; Godefroid and Jagadeesan 2002;
Godefroid 2014), and implemented in Yasm (Gurfinkel et al.
2010). More specific vari-
ty

Last but not least, most of the existing works have been
defined only theoretically (with the exceptions mentioned
above), and their usability has never been considered from
the perspective of a user with no intimate knowledge of veri-
fication techniques.

3 Preliminaries

We start by introducing the models and formulae which serve
as an input to model checking.

3.1 MAS Graphs

To represent the behaviour of a multi-agent system, we use
modal representations inspired by reactive modules (Alur
and Henzinger 1999), interleaved interpreted systems (Lo-
muscio et al. 2010a; Jamroga et al. 2020a), and in particu-
lar by the way distributed systems are modelled in UP-
PAAL (Behrmann et al. 2004).

Let $\text{Var}$ be a finite set of typed variables over finite do-

cains. By $\text{Eval}(\text{Var})$ we denote a set of evaluations, i.e.,
functions mapping variables $v \in \text{Var}$ to values from their
domains $\text{dom}(v)$. $\text{Cond}$ is a set of logical conditions (also
called guards) over $\text{Var}$, possibly involving arithmetic op-
erators. Let $\text{Chan}$ be a finite set of asymmetric one-to-one
synchronization channels. We define the set of synchroniza-
tions as $\text{Sync} = \{c! , c? | c \in \text{Chan}\} \cup \{-\}$, with $c!$ and $c?$
for sending and receiving on a channel $c$, respectively, and
“--” for no synchronization.

Definition 1 (Agent graph). An agent graph is a tuple $G =
(\text{Loc}, \text{Var}, l_0, \{g\}, \text{Act}, \text{Effect}, \rightarrow)$, consisting of:

- $\text{Loc}$: a non-empty finite set of locations;
- $\text{Var}$: a finite set of typed variables over finite domains;
- $l_0 \in \text{Loc}$: the initial location;
- $g_0 \in \text{Cond}$: the initial condition;
- $\text{Act}$: a set of actions, with $\tau \in \text{Act}$ for “do nothing”;
- $\text{Effect} : \text{Eval}(\text{Var}) \times \text{Act} \rightarrow \text{Eval}(\text{Var})$: the effect of
an action. We assume $\text{Effect}(\eta, \tau) = \eta$;
- $\rightarrow \subseteq \text{Loc} \times \text{Label} \times \text{Loc}$: a set of labelled edges with
labels from $\text{Label} \subseteq \text{Cond} \times \text{Sync} \times \text{Act}$, which will be
used to define the local transition relation.

Instead of $(l, \text{labl}, l') \in \rightarrow$, we will often write $l \overset{g \cdot \text{ch} \alpha}{\rightarrow} l'$,
where $g = \text{cond}(\text{labl})$, $\text{ch} = \text{sync}(\text{labl})$ and $\alpha = \text{act}(\text{labl})$
Also, we will omit $\text{labl} = --$.

Each condition $g \in \text{Cond}$ can be associated with its set
of satisfying evaluations $\text{Sat}(g) = \{\eta \in \text{Eval}(\text{Var}) | \eta \models g\}$. An
dge labelled by $\text{labl} \in \text{Label}$ is locally enabled for evaluation $\eta \in \text{Eval}(\text{Var})$ if $\eta \models \text{cond}(\text{labl})$. For simplicity, we assume that $\text{Sat}(g_0) = \{g_0\}$, i.e., each variable $v \in \text{Var}$
is initialized by its default value $v_0 = g_0(v)$.

Furthermore, every action $\alpha \in \text{Act} \setminus \{\tau\}$ can be associ-
ated with a non-empty sequence of atomic assignments (also
called updates) of the form $\alpha^{(1)} \alpha^{(2)} \ldots \alpha^{(m)}$.

Without loss of generality, we assume that the variables in
$\text{Var} = \{v_1, \ldots, v_k\}$ are ordered in an arbitrary way.
Thus, the evaluation of $V \subseteq \text{Var}$ can be seen as a vector

---

2(Cohen et al. 2009; Belardinelli et al. 2019) use modular rep-
resentations of the concrete state space, but they do need a global
representation of the concrete transition space, and they generate
the global abstract model explicitly.

3We consider only variables with finite domains, in line with
most model checking algorithms and tools for MAS.
We define the execution of a MAS graph by its unwrapping.

**Definition 3** (Combined MAS graph). Let $MG = \{\langle Var, G^1, \ldots, G^n \rangle \}$ be a MAS graph having a set of shared variables $Var$. The combined MAS graph of $MG$ is the agent graph $G_{MG} = (\langle Loc, V, l_0, g_0, \text{Act}, \text{Effect}, \rightarrow \rangle)$, where $Var = \bigcup_{i=1}^{n} Var_i$, $Loc = \bigcup_{i=1}^{n} Loc^i \times \ldots \times Loc^i$, $l_0 = (l_0^1, \ldots, l_0^n)$, $g_0 = g_0^1 \land \ldots \land g_0^n$, $\text{Act} = \bigcup_{i=1}^{n} \text{Act}^i$.

Relation $\rightarrow$ is obtained inductively by the following rules (where $l_i, l'_i \in Loc^i$, $l_j, l'_j \in Loc^j$, $c \in Chan^i \cap Chan^j$ for two agent graphs $G^i$ and $G^j$ of distinct indices $1 \leq i, j \leq n$):

\begin{align*}
l_i \xrightarrow{\alpha, \eta} l'_i & \quad \text{if } \alpha \in Act^i \\
& \quad \text{and } \eta \in \text{Eval}^i(l_i) \\
(l_i, l_j) \xrightarrow{\eta \land \text{Cond}^i} (l'_i, l'_j) & \quad \text{if } \alpha \in Act^j \\
& \quad \text{and } \eta \in \text{Eval}^j(l_i) \\
(l_i, l_j) \xrightarrow{\eta \land \text{Cond}^i} (l'_i, l'_j) & \quad \text{if } \alpha \in Act^j \\
& \quad \text{and } \eta \in \text{Eval}^j(l'_i)
\end{align*}

Lastly, the effect function is defined by:

$$\text{Effect}(\alpha, \eta) = \begin{cases} \text{Effect}^i(\alpha, \eta) & \text{if } \alpha \in Act^i \\
\text{Effect}(\alpha, \text{Effect}(\alpha, \eta)) & \text{if } \alpha = \alpha \circ \alpha_j \end{cases}$$

**Example 2.** The combined MAS graph $G_{ASV}$ for asynchronous simple voting of Example 1 is depicted in Fig. 2a.

Intuitively, the combined MAS graph is an asynchronous composition of the agent graphs in $MG$. Note that by the construction of combined MAS graph, its edges are always labelled by $\text{labl} \in \text{Label}$, s.t. $\text{sync}(\text{labl}) = \top$. To turn it into a model, we still need to instantiate the variables in combined MAS graph with their possible values.

**Definition 4** (Model). A model is a tuple $M = (\mathcal{S}, I, \rightarrow, AP, L)$, where $\mathcal{S}$ is a set of states, $I \subseteq \mathcal{S}$ is a non-empty set of initial states, $\rightarrow \subseteq \mathcal{S} \times \mathcal{S}$ is a transition relation, $AP$ is a set of atomic propositions, $L : \mathcal{S} \rightarrow 2^{AP}$ is a labelling function. We assume $\rightarrow$ to be serial, i.e., there is at least one outgoing transition at every state. We also assume that $I$ includes only states reachable from $I$.

Nodes and edges in an agent graph $G$ correspond to sets of states and transitions, defined by the unwrapping of $G$.

**Definition 5** (Unwrapping). The unwrapping of an agent graph $G$ is a model $\mathcal{M}(G) = (\mathcal{S}, I, \rightarrow, AP, L)$, where:

- $\mathcal{S} = \mathcal{L} \times \text{Eval}(\mathcal{Var})$
- $I = \{ (l_0, \eta) \in \mathcal{S} \mid \eta \in \text{Sat}(g_0) \}$
- $\rightarrow = \rightarrow_0 \cup \{ (s, s') \in \mathcal{S} \times \mathcal{S} \mid \exists \tilde{s}' \in \mathcal{S} . s \rightarrow_{0} \tilde{s}' \}$,
- $\text{where } \rightarrow_0 = \{ (\langle \ell . \eta, \ell' . \eta' \rangle) \in \mathcal{S} \times \mathcal{S} \mid \exists l \xrightarrow{\alpha_0} \ell' . \eta \in \text{Sat}(g) \land \eta' = \text{Effect}(\alpha, \eta) \}$
- $AP = \mathcal{L} \cup \text{Cond}$
- $L(\langle \ell . \eta \rangle) = \{ l \} \cup \{ g \in \text{Cond} \mid \eta \in \text{Sat}(g) \}$

---

\(^3\)Add loops wherever necessary to make the relation serial.

---

**Example 1 (ASV).** As the running example, we use a variation of the Asynchronous Simple Voting scenario of (Jamroga et al. 2020a). Its MAS graph $ASV = \{\langle Var, G^{\text{Voter}}, G^{\text{Coercer}} \rangle\}$ is shown in Fig. 1. The system is parameterized by the number of candidates $NC$.

The voter starts by nondeterministically selecting one of the candidates (idle → voted), for whom the vote will be cast. Then, she decides to either give the proof of how she voted to the coercer (voted → obeyed), or to refuse it (voted → disobeyed). Both options require executing a synchronous transition (using channels $g$ and $ng$) with the coerer. In turn, the coerer either gets the proof and learns for whom the vote was cast, or becomes aware of the voter’s refusal.
The unwrapping $M(MG)$ of a MAS graph $MG$ is given by the unwrapping of its combined graph.

Intuitively, each state in the unwrapping specifies a location in the MAS graph plus a tuple of values for all the variables. Moreover, the atomic statements in $AP$ allows us to indicate a location, or refer to a Boolean condition. By $AP(V)$, we will denote the subset of propositions that do not use any variables from outside $V$.

Example 3. The unwrapping of the MAS graph for asynchronous simple voting with 3 candidates is shown in Fig. 3.

Definition 6 (Runs, paths, local domain). Let $M$ be a model. A run in $M$ is a sequence of states $s_0 s_1 \ldots$, such that $s_i \in St$ and $s_i \rightarrow s_{i+1}$ for every $i$. For a finite run $\pi = s_0 s_1 \ldots s_n$, let $\text{len}(\pi) = n$ denote its length. By $\pi[i]$ and $\pi[i,j]$ we denote the $k$-th state of $\pi$ and the fragment of $\pi$ from index $i$ to $j$. A path is an infinite run. The sets of all runs in $M$, all paths in $M$, and all paths starting from state $s$ are denoted by $\text{Runs}(M)$, $\text{Paths}(M)$, and $\text{Paths}(s)$. Similarly, $\text{Runs}$ denotes the set of runs of fixed length $t \in \mathbb{N}^+$.

A local domain is a function $d : \text{Loc} \rightarrow \mathcal{P}(\text{Eval}(\text{Var}))$ that maps each location to the set of evaluations reachable at $l$ (i.e., for which there exists a corresponding state in the model). By $d(l)(V) = \{\eta(V) \mid \eta \in d(l)\}$ we denote the restriction of $d(l)$ that considers only the values of $V \subseteq \text{Var}$.

3.3 Branching-Time Logic $\text{ACTL}^*$

To specify requirements, we use the universal fragment of the branching-time logic $\text{CTL}^*$ (Emerson 1990), denoted $\text{ACTL}^*$ with $A$ (“for every path”) as the only path quantifier. The syntax for $\text{ACTL}^*$ over a set of atomic propositions $AP$ is formally given by:

$$\psi ::= T \mid \neg \psi \mid \psi \lor \psi \mid \phi \lor \psi \mid \exists a \phi$$

$$\phi ::= \psi \mid \phi \land \phi \mid \phi \lor \phi \mid X \phi \mid [a] \phi$$

where $a \in AP$, and $X, U, R$ stand for “next”, “until” and “release” respectively. Formulae $\phi$ are called state formulae, and $\psi$ are called path formulae. The semantics of $\text{ACTL}^*$ is given with respect to states $s$ and paths $\pi$ of a model $M$.

$$M, s \models a \iff a \in L(s)$$

$$M, s \models A\phi$$

$$M, s \models [a] \phi \iff a \in L(s)$$

$$M, s \models X\phi \iff \exists i, j \leq n \exists s' \in s[I] . s' \models \phi$$

$$M, s \models [a \land \phi] \psi \iff M, s \models [a \land \phi] \psi$$

$$M, s \models [a \lor \phi] \psi \iff M, s \models [a \lor \phi] \psi$$

The clauses for Boolean connectives are standard. Additional temporal operators “sometime” and “always” can be defined $\text{MG}$.

Example 4. Model $M = M(ASV)$ in Fig. 3 satisfies the $\text{ACTL}^*$ formula $A\neg \text{disobeyed} \land \text{Kvt} = 0$, saying that if Voter obeys, Coercer gets to know how she voted, and the formula $A\neg \text{disobeyed} \land \text{Kvt} = 0$, expressing that she cannot disobey Coercer’s instructions without his knowledge. It does not satisfy $A\neg \text{Kvt} \land \neg \text{Kvt} = 0$, saying that Coercer will eventually get to know how Voter voted.

4 Variable Abstraction for MAS Graphs

In this section, we propose how to automatically reduce MAS graphs by simplifying their structure of local variables. As the starting point, we take the idea of may/must abstractions (Dams et al. 1997; Godefroid and Jagadeesan 2002). Typically, they take concrete states and cluster them according to a given equivalence relation. The $\text{may}$ model includes transitions of type $\exists I, \text{Var}$. Similarly, $\text{must}$ model includes transitions of type $\forall I, \text{Var}$. $\text{may}$ and $\text{must}$ model are related by $\text{may} \leq \text{must}$.

In our case, concrete states are pairs $(l, \eta)$. Arguably the simplest equivalence is given by removing a subset of variables $V'. I$, that is, we will cluster states $(l_1, \eta_1)$ and $(l_2, \eta_2)$ iff $l_1 = l_2$ and $\eta_1 = \eta_2$ agree on the variables in $\text{Var} \setminus V'$.

Moreover, we want the abstraction $A$ to transform the MAS graph $MG = (\text{Var}, G^1, \ldots, G^n)$ so that:

(i) computation of the abstraction is agent-based, i.e.,

$$A(MG) = (\text{Var}, A(G^1), \ldots, A(G^n))$$

(ii) the abstract agent graphs $A(G^i)$ have the same structure of locations as their concrete versions $G^i$;

(iii) the only change results from removal of a subset of local variables $V$, or simplifying their domains of values.

The may-abstraction $A^{\text{may}}(MG)$ should over-approximate $MG$, in the sense that every transition in $MG$ has its counterpart in $A^{\text{may}}(MG)$. Consequently, every formula of type $\phi$ that holds in the model $A^{\text{may}}(MG)$ also must hold in the model $A(MG)$. Likewise, the must-abstraction $A^{\text{must}}(MG)$ should under-approximate $MG$, in the sense that all transitions in $A^{\text{must}}(MG)$ have their counterparts in $MG$. Thus,
whenever \( A\varphi \) is false in \( \mathcal{M}(A^{\text{equt}}(MG)) \), it is also false in \( \mathcal{M}(MG) \).

The general structure of the procedure is shown in Alg. 1. First, we approximate the set of reachable evaluations \( d(l)\) for every \( l \in \text{Loc} \) in every location of the combined MAS graph \( G_{MG} \) by means of Alg. 2, discussed in Section 4.2. Then, the output is used to transform the agent graphs \( G^i \) in \( MG \), one by one, by detecting and transforming the occurrences of the variables in \( V \cap \text{Var}^i \). This is implemented by function ComputeAbstraction (Algorithm 3), which will be presented in detail in Sections 4.3–4.5.

### 4.2 Approximating the Domains of Variables

Given MAS graph \( MG \), the approximation of reachable values for a set of variables \( V \subseteq \text{Var} \) is defined in two variants. The upper-approximation of local domain (denoted \( d^+ \)) for every \( l \in \text{Loc} \) initializes \( d^+(l) \) to \( \emptyset \), and then adds new, possibly reachable values of \( l \) whenever they are produced on an edge coming to \( l \). The lower-approximation (denoted \( d^- \)) initializes \( d^-(l) = \text{dom}(V) \), and iteratively removes the values might be unreachable. To this end, function ApproxLocalDomain is parameterized by symbols \( d_0 \) and \( \emptyset \), such that \( d_0 = \emptyset \) and \( \emptyset \) for the upper-approximation, and \( d_0 = \text{dom}(V) \) and \( \emptyset = \cap \) for the lower-approximation. Note that \( d_0 \) is simply a neutral element of the operation \( \cap \).

Furthermore, for an approximation of local domain \( d^* \), where \( * \in \{+, -\} \), defined on \( \text{Loc} = \text{Loc}^1 \times \ldots \times \text{Loc}^n \), by \( d^*_i \) we denote a reduced to the \( i \)-th location component “narrowing” of that, where \( 1 \leq i \leq n \). Intuitively, for \( l_j \in \text{Loc} \) the value of \( d^*_i(l_j) \) is defined as \( \emptyset \) and the value of \( d^*_i(l_j) \) is defined as \( \emptyset \times \text{Loc}^{i-1} \times \ldots \times \text{Loc}^{n} \) for \( * \in \{+, -\} \) with each visit at \( l \). This proceeds until a stable approximation is obtained. Each location \( l \) must be visited at least once, and whenever some of its predecessors \( l' \) get their approximations \( d^*(l') \) refined, the location \( l \) must be processed again.

The max-priority queue \( Q \) stores the locations that must be visited (possibly anew). Within the queue, the higher traversal priority is given to locations with greater reachability index \( r(l) \), defined as the number of locations \( l' \neq l \) reachable from \( l \). This will reduce the number of potential re-visits in comparison with the generic FIFO variant.

### Algorithm 1: Abstraction of MAS graph \( MG \) wrt \( V \)

```
for \( MG = \{\{\text{Var}_i, G^i, \ldots, G^n\}\} \) compute the combined graph \( G_{MG} \)
2. compute the approximate local domain \( d \) for \( V \) in \( G_{MG} \)
3. foreach agent graph \( G^i \in MG \) do
4. \hspace{1em} compute abstract graph \( A(G^i) \) wrt. \( d_i \)
5. return \( A(MG) = \{\{\text{Var}_i, A(G^i), \ldots, A(G^n)\}\} \)
```

### Algorithm 2: Approximation of local domain for \( V \subseteq \text{Var} \)

```
ApproxLocalDomain(G = G_{MG}, V)
1. foreach \( l \in \text{Loc} \) do
2. \hspace{1em} \( l.d := d_0 \)
3. \hspace{1em} \( l.p := \emptyset \)
4. \hspace{1em} \( l.color := \text{white} \)
5. \hspace{1em} \( l.d := (\eta(V) | \eta \in \text{Sat}(g_0)) \)
6. \hspace{1em} \( Q := \emptyset \)
7. Enqueue(Q, l_0)
8. while \( Q \neq \emptyset \) do
9. \hspace{1em} \( l := \text{ExtractMax}(Q) \)
10. VisitLoc(l, V)
11. if \( l.color \neq \text{black} \) then
12. \hspace{1em} foreach \( l' \in \text{Succ}^{-1}(l) \) do
13. \hspace{2em} \( Q := \text{Enqueue}(Q, l') \)
14. \hspace{2em} \( l'.p := l'.p | \{l\} \)
15. \hspace{1em} \( l.color = \text{grey} \)
16. return \( \{(l, V, l.d) | l \in \text{Loc} \} \)
```

The algorithm associates with each location \( l \) its attributes \( l.color \in \{\text{white}, \text{grey}, \text{black}\} \), the set of relevant predecessors \( l.p \subseteq \text{Loc} \setminus \{\} \), and the current approximation of the local domain \( l.d \). The colour indicates if the location has not been visited yet (white), its \( l.d \) has been refined (grey), or it has been visited and closed (black). The set \( l.p \) indicates which predecessors of \( l \) had their approximations updated, which may lead to a refined \( l.d \).

In lines 1–5, the locations are initialized with \( \text{white} \), the empty set of predecessors, and the initial approximation \( d_0 \). Lines 6–7 initialize the queue with location \( l_0 \). The while-loop of lines 7–15 describes the visit in location \( l \). In VisitLoc, after the edges from \( l.p \) were taken into account for \( l.d \), the \( l.p \) is reset (line 20). Self-loops are processed until \( l.d \) stabilizes (lines 23–28). The function ProcEdge explores the possible transitions, and gradually computes the image (restricted by \( V \)) associated with updates from \( \alpha \) on evaluations satisfying the guard \( g \) and having their \( V \) counterpart in \( l.d \). Lastly, if \( l \) changes its colour to grey from either black or
Table 1: Reachability index r of locations and reachable values of vt from lower-approximation d−, exact local domain d and upper-approximation d+ in ASV with 3 candidates

<table>
<thead>
<tr>
<th>Location</th>
<th>r(ℓ)</th>
<th>d−(ℓ)</th>
<th>vir</th>
<th>d(ℓ)</th>
<th>vir</th>
<th>d+(ℓ)</th>
<th>vir</th>
</tr>
</thead>
<tbody>
<tr>
<td>⟨idle, idle⟩</td>
<td>3</td>
<td>{0}</td>
<td>{0}</td>
<td>{0}</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>⟨voted, idle⟩</td>
<td>2</td>
<td>∅</td>
<td>{1, 2, 3}</td>
<td>{1, 2, 3}</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>⟨obeyed, halt⟩</td>
<td>0</td>
<td>∅</td>
<td>{1, 2, 3}</td>
<td>{1, 2, 3}</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>⟨disobeyed, halt⟩</td>
<td>0</td>
<td>∅</td>
<td>{1, 2, 3}</td>
<td>{1, 2, 3}</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Algorithm 3: Abstraction by variable removal

1. ComputeAbstraction(G = G′, V, d = d₀)
   2. ch = 0
   3. foreach ℓ ∈ d(ℓ) do
   4.     g′ = g[V = c]
   5.     δ₀ = {η ∈ Sat(g) | η(V) = c}
   6.     α′ = α(1) . . . α(春)
   7.     foreach i = 1 to m do
   8.         δᵢ = {η′ = Effect(η, α(ⁱ)) | η ∈ δᵢ₋₁}
   9.         if lhs(α(ⁱ)) ∈ V then
   10.            δᵢ = δᵢ ∪ {η ∈ δᵢ | η(V) = α(ⁱ)(V)}
   11.            δᵢ = δᵢ ∪ {η ∈ δᵢ | η(V) = α(ⁱ)(V)}
   12.            α′ = α′ ∪ g′(ch α′ → ℓ’)
   13.            ch := ch + 1
   14.     g₀ = g₀[V = η₀(V)]
   15.     Var′ := Var′ \ V
   16.     return G

white, then all the immediate neighbours are enqueued to be inspected, adding l to the list of their relevant predecessors, and changing its colour to black (lines 10–15).

The algorithm halts and returns a stable approximation d (line 16) when the queue is empty and all the locations are black. It runs in polynomial time w.r.t. the number of locations and joint valuations of the removed variables. Note that the subsequent approximations l, d are weakly monotonic (i.e., l, d ⊆ l, d for d+, and l, d ⊇ l, d for d−). Since the sets of locations and edges are finite, and so are the variable domains, termination is guaranteed.

Example 5. The local domain and its approximations obtained by ApproxLocalDomain for variable vt in the combined ASV graph of Example 2 can be found in Tab. 1.

4.3 Abstraction by Removal of Variables

The simplest form of abstraction consists in the complete removal of a given subset of variables V ⊆ Var from the MAS graph. To this end, we use the approximation of reachable values of V, produced by ApproxLocalDomain. More precisely, we transform every edge between l and ℓ′ that includes variables V ⊆ V in its guard and/or its update into a set of edges (between the same locations), each obtained by substituting V with a different value C ∈ d(l)|vir, see Alg. 3. The abstract agent graph obtained by removing variables V from G in the context of MG is denoted by A_{\{x\}}(G, MG).

4.4 Merging Variables and Their Values

A more general variant of variable abstraction assumes a collection of mappings F = {f₁, . . . , fₘ}. Each mapping fᵢ : Eval(Xᵢ) → Eval(zᵢ) merges the local variables Xᵢ ⊆ Var¹ of some agent graph G¹ to a fresh variable zᵢ. The abstraction based on fᵢ removes variables Xᵢ from graph G¹, and replaces them with zᵢ that “clusters” the values of Xᵢ into appropriate abstraction classes. We will use ArgP₁(fᵢ) = Xᵢ and ArgP₂(F) = ∪ₘ ArgP₂(fᵢ) to refer to the variables removed by fᵢ and F. ArgP₁(fᵢ) = {zᵢ} and ArgP₂(F) = ∪ₘ ArgP₂(fᵢ) refer to the new variables.

Note that the procedure in Section 4.3 can be seen as a special case, with a sole mapping f merging V to a fresh variable z with the singleton domain dom(z) = {η₀(z)}.

4.5 Restricting the Scope of Abstraction

The abstraction scheme can be further generalised by considering a set of mappings F = {(f₁, Sc₁), . . . , (fₘ, Scₘ)}, with each fᵢ : Eval(Xᵢ) → Eval(zᵢ) applied in some agent graph G¹, and Scᵢ ⊆ Loc¹ defining the scope of fᵢ. That is, mapping fᵢ is applied only in the locations l ∈ Scᵢ by assigning fᵢ(χᵢ) to zᵢ, and resetting the value of each v ∈ Xᵢ to v₀. Outside of Scᵢ, the variables in Xᵢ stay intact, and the new variable zᵢ is assigned an arbitrary default value.

The abstract agent graph obtained by function ComputeAbstraction from G in the context of MG via F is denoted by A_F(G, MG). Consequently, the abstraction of MG = [{Varsh, A_F(G¹)}] becomes A_F(MG) = [{Varsh, A_F(G¹), . . . , A_F(G^n, MG)}].
The general algorithm is presented in detail in Section A.2 of the supplementary material (https://tinyurl.com/3eukrkrkb).

5 Correctness of Variable Abstraction

We will now prove that the abstraction scheme preserves the truth values of ACTL\* formulae if the computation of variable domain \( d \) produces the right approximation of their reachable values. In essence, we show that the abstraction always produces an approximation of the runs in the concrete MAS graph, which induces an appropriate simulation relation, and thus guarantees (one-way) preservation of ACTL\*.

5.1 Simulations between Models

We first recall a notion of simulation between models (Baier and Katoen 2008; Clarke et al. 2018; Cohen et al. 2009).

Definition 7. Let \( M_i = (S_i, I_i, \rightarrow_i, P_i, L_i) \), \( i = 1, 2 \) be a pair of models, and let \( AP \subseteq AP_1 \cap AP_2 \) be a subset of atomic propositions. Model \( M_2 \) simulates model \( M_1 \) over \( AP \) (written \( M_1 \preceq AP M_2 \)) if there exists a simulation relation \( \mathcal{R} \subseteq S_1 \times S_2 \) over \( AP \), such that:

(i) for every \( s_1 \in I_1 \), there exists \( s_2 \in I_2 \) with \( s_1 \mathcal{R} s_2 \);

(ii) for each \( (s_1, s_2) \in \mathcal{R} \):

(a) \( L_1(s_1) \cap AP = L_2(s_2) \cap AP \), and

(b) if \( s_1 \rightarrow_i s'_1 \) then there is \( s_2 \rightarrow_i s'_2 \) such that \( s_1 \mathcal{R} s'_2 \).

Additionally, for a pair of reachable states \( s_1, s_2 \) in \( M_1, M_2 \) such that \( (s_1, s_2) \in \mathcal{R} \), we say that the pointed model \( (M_2, s_2) \) simulates \( (M_1, s_1) \) over \( AP \), and denote it by \( (M_1, s_1) \preceq AP (M_2, s_2) \).

Theorem 1. For \( (M_1, s_1) \preceq AP (M_2, s_2) \) and any \( \text{ACTL}^* \) state formula \( \psi \), built of propositions from \( AP \) only, it holds that:

\[ M_2, s_2 \models \psi \implies M_1, s_1 \models \psi \quad (\ast) \]

The proof is standard, see e.g. (Baier and Katoen 2008).

Remark. In our abstraction scheme, the set of joint atomic propositions \( AP \), underlying the simulation relation, consists of Boolean conditions and a subset of variables that are not removed from the MAS graph.

5.2 May-Abstractions of MAS Graphs

Let \( M_1 = \mathcal{M}(MG_1), M_2 = \mathcal{M}(MG_2) \) be models resulting from unwrapping of MAS graphs \( MG_1, MG_2 \). We start with a notion of correspondence between states and runs. Then, we use it to define the concept of may-approximation. The following is straightforward.

Lemma 2. Let \( V \subseteq \text{Var} \) and \( V' = \text{Var} \setminus V \), \( \alpha \in \text{Act} \) and \( \text{Effect}(\alpha, \eta_1) = \eta'_1 \). For any \( \eta_2 \in \text{Eval}(\text{Var}) \), we have:

\[ \eta_1 \models \psi \quad \eta_2 \not\models \psi \implies \text{Effect}(\alpha[V' = \eta_1(V')]) \models \psi \quad \eta_1 = \psi \eta'_2 \]

Definition 8. Let \( s_i \in S_i \) and \( s_i = (l_i, \eta_i) \) for \( i = 1, 2 \). State \( s_2 \) corresponds to a state \( s_1 \) over variables \( V \subseteq \text{Var}_i \cap \text{Var}_2 \) (denoted \( s_1 \models_{V, s_2} s_2 \)) iff \( l_1 = l_2 \) and \( \eta_1 = \psi \eta_2 \).

Moreover, run \( \pi_2 \in \text{Runs}(M_2) \) corresponds to run \( \pi_1 \in \text{Runs}(M_1) \) with respect to \( V \) (denoted \( \pi_1 \models_{V, \pi_2} \pi_2 \)) iff:

(i) \( \text{len}(\pi_1) = \text{len}(\pi_2) = t \), and

(ii) for every \( 1 \leq i \leq t \), it holds that \( \pi_1[i] \models \psi \pi_2[i] \).

5.3 Variable Abstraction Is Sound

We prove now that the abstraction method, based on upper-approximation of local domain, is indeed a simulation.

Lemma 3. Let \( MG \) be a MAS graph and \( d^+ \) be an upper-approximation of a local domain defined on \( V \subseteq \text{Var} \). Then, for any state \( \langle l, \eta \rangle \in \mathcal{M}(MG) \), it must be that \( \eta(W) \in d^+(l) \) for any \( W \subseteq V \).

Remark. if \( g \in \text{Cond} \) and \( \text{Sat}(g) = \{ \eta_1, \ldots, \eta_k \} \), then \( g \models \bigvee_{1 \leq i \leq k} \bigwedge_{v \in \text{Var}} (v = \psi(v)) \).

Let \( MG = (\text{Var}_k, G_1^1, \ldots, G_n^n) \), where \( G^i = (\text{Var}_i, \text{Loc}_i, l_0^i, g_0^i, \text{Act}_i, \text{Effect}_i, \rightarrow_i) \), \( MG = (\text{Var}_1, \text{Loc}_1, l_0, \text{Act}, \text{Effect}, \rightarrow) \), \( MG = \mathcal{A}^\text{may}(MG) \), \( G^i_\mathcal{E} = (\text{Var}_2, \text{Loc}_0, g_0, \text{Act}, \text{Effect}, \cdots) \).

Theorem 4. Let \( M_1 = \mathcal{M}(MG) \) and \( M_2 = \mathcal{M} (\mathcal{A}^\text{may}(MG)) \), s.t.

\[ M_i = (S_i, I_i, \rightarrow_i, P_i, L_i) \text{ for } i = 1, 2 \]

\( \mathcal{V} \subseteq \text{Var}_1 \cap \text{Var}_2 \). Then, a relation \( \mathcal{R} \subseteq S_1 \times S_2 \) where \( \langle l_1, \eta_1 \rangle \mathcal{R} \langle l_2, \eta_2 \rangle \) iff \( l_1 = l_2 \text{ and } \eta_1 \models \psi \eta_2 \), is a simulation relation over \( AP = AP_1 \cap AP_2 \) between \( M_1 \) and \( M_2 \).

Proof. Here, we will present a proof for a simpler case - variable removal; proof for a general case is only technically more involved and can be found in supplementary material.

Recall that for \( \text{Sat}(g_0) = \{ \eta_0 \} \) it holds \( g_0 \equiv \bigwedge_{v \in \text{Var}_1} v = \psi(v) \). In variable removal scenario \( Var_2 = Var_1 \setminus V \) and \( Var_2 = Var_2 \). Therefore \( g_0 \equiv \bigwedge_{v \in V} v = \psi(v) \). From this and the fact that the sets of locations for \( MG \) and \( \mathcal{A}^\text{may}(MG) \) are the same, we can conclude that the condition (i) of Definition 7 must hold.

Now we show that condition (ii) of Definition 7 holds as well. By construction, each concrete \( (l, labl, l') \in \mathcal{M} \) from \( MG \) will have (at least one) matching abstract edge \( (l, labl, l') \in \rightarrow \), where \( labl = labl[V = c] \) for some \( c \in d^+(l') \). Therefore, for any \( \langle l_1, \eta_1 \rangle \mathcal{R} \langle l_2, \eta_2 \rangle \) and \( \langle l_1, \eta_1 \rangle \rightarrow l' \), \( l' \in \mathcal{M} \) that was induced by an edge \( (l, labl_1, l') \in \rightarrow \), where \( \eta_1 = \text{cond}(labl_1) \). From this and the fact that the sets of locations for \( MG \) and \( \mathcal{A}^\text{may}(MG) \) are the same, we can conclude that the condition (ii) of Definition 7 must hold.

We can now state our main theoretical result.

Theorem 5. Let \( MG \) be a MAS graph, and \( F \) a set of mappings as defined in Section 4.5. Then, for every formula \( \psi \) of \( \text{ACTL}^* \) that includes no variables being removed or added by \( F \):

\[ \mathcal{M}(\mathcal{A}^\text{may}(MG)) \models \psi \implies \mathcal{M}(MG) \models \psi. \]

Proof. Follows directly from Theorems 1 and 4. \( \square \)
5.4 Must-Abstractions of MAS Graphs

An analogous result can be obtained for must-abstraction.

**Lemma 6.** Let $MG$ be a MAS graph and $d^-$ be a lower-approximation of a local domain defined for $V \subseteq \text{Var}$. By the very nature of $d^-$, for any reachable location $l \in \text{Loc}$ it can have at most one element $[d^-(l)]_v \leq 1$. Moreover, when $d^-(l)]_v = \{c\}$ there must exist reachable in $M(MG)$ state $(l, \eta)$, where $\eta(V) = c$.

**Theorem 7.** Let $M_1 = M(MG)$ and $M_2 = M(A^\text{must}_F(MG))$, s.t. $M_i = (S_i, I_i, \longrightarrow_i, AP_i, L_i)$ for $i = 1, 2, V = \text{Args}_R(F)$, $\overline{\eta} \subseteq \text{Var}_1 \cap \text{Var}_2$. Then, a relation $R \subseteq S_2 \times S_1$, where $\langle l_2, \eta_2 \rangle R \langle l_1, \eta_1 \rangle$, iff $l_2 = l_1 \wedge \eta_2 = \overline{\eta}_1$, is a simulation relation over $AP = AP_1(\overline{\eta}_1) \cap AP_2(\overline{\eta}_2)$ between $M_2$ and $M_1$.

The proof is analogous to that of Theorem 4, see the supplementary material for details.

**Theorem 8.** For each formula $\psi \in \text{ACTL}^*$ including no variables removed by $F$: $M(A^\text{must}_F(MG)) \models \psi$ implies $M(MG) \models \psi$.

5.5 Abstraction on MAS Templates

When some agent graphs in the MAS graph are instantiations of a single template, one can apply abstraction directly on the template. This typically results in a coarser abstraction of the original MAS graph, but such abstractions are exponentially faster to compute, as the size of the model underlying the MAS graph is exponential in the size of the agent template.

**Definition 9.** (MAS template). A MAS template is a compact representation of a MAS graph $MG$ as a tuple $MT = (\text{Var}_{sh}, \text{Const}_{sh}, (GT^1, \#^1), \ldots, (GT^k, \#^k))$ which lists pairs of agent templates $GT^i$ and the number of their instances $\#^i$ in $MG$, as well as the sets of shared variables $\text{Var}_{sh}$ and shared constants $\text{Const}_{sh}$.

An agent template $GT^i$ is just an agent graph, instantiated in $MG$ by $\#^i$ copies through adding their id’s $j = 1, \ldots, \#^i$ as prefixes to the locations and local variables in $GT^i$.

In order to avoid unfolding the MAS template into a MAS graph, we approximate the potential synchronization between instances of agent templates when doing abstraction. More precisely, the upper-approximation of a local domain $d_i$ in agent template $GT^i$ is computed on $\text{upsync}(GT^i)$ that discards all the synchronisation labels from the edges in $GT^i$. Analogously, the lower-approximation of a local domain $d_i$ in agent template $GT^i$ is computed on $\text{lowsync}(GT^i)$ that discards all the edges with synchronisation labels from $GT^i$.

**Theorem 9.** Let $MT$ be a MAS template, corresponding to the MAS graph $MG$. Then $A^\text{may}(\text{upsync}(MT))$ induces a may-abstraction of $MG$, and $A^\text{must}(\text{lowsync}(MT))$ induces a must-abstraction of $MG$.

**Proof.** Follows directly from the fact that discarding synchronisation labels results in a coarser upper-approximation of the local domain, and discarding the edges with synchronisation labels results in a coarser lower-approximation of $d_i$.

6 Case Study and Experimental Results

We evaluate our abstraction scheme on a real-life scenario.

6.1 Case Study: Integrity of Postal Voting

As input, we use a scalable family of MAS graphs that specify a simplified postal voting system. The system consists of a single agent graph for the Election Authority (depicted in Fig. 5a) and $N V$ instances of eligible Voters (Fig. 5b).

Each voter can vote for one of the $N C$ candidates. The voter starts at the location $\text{Id}le$, and declares if she wants to receive the election package with the voting declaration and the ballot by post, or to pick it up in person. Then, the voter waits until the package can be collected, which leads to location $\text{has}$. At that point, she sends the forms back to the authority, either filled in or blank (e.g., by mistake). The authority collects the voters’ intentions (at location $\text{coll_dec}$), distributes the packages (at $\text{send_ep}$), collects the votes, and computes the tally (at $\text{coll_vts}$). A vote is added to the tally only if the declaration is signed and the ballot is filled.

In the experiments, we verify the formula $\varphi_{\text{ballot}} \equiv \text{AG}([\sum_{i=1}^{NC} \text{tally}[i] \leq \sum_{j=1}^{NV} \text{pack_sent}[j] \leq NV] \text{expressing a variant of resistance to ballot stuffing}. More precisely, the formula says that the amount of sent packages can never be higher than the number of voters, and there will be no more tallied votes than packages. The formula is satisfied in all considered instances of our voting model.

Due to space limitations, we only present results for may-abstraction — arguably, the more important case, since it can be used to prove an $\text{ACTL}^*$ formula true in a model. Experimental results for must-abstraction are shown in the supplementary material.

6.2 Results of Experiments

We have used the following abstractions:

- Abstraction 1: globally removes variables $mem_{sg}$ and $mem_{vt}$, i.e., the voters’ memory of the cast vote and whether the voting declaration has been signed;
- Abstraction 2: removes the voter’s memory of her decision (variable $\text{mem}_{dec}$) at locations $\{\text{has}, \text{voted}\}$, and $\text{dec_recv}$ at $\{\text{coll_vts}\}$;
- Abstraction 3: combines Abstractions 1 and 2.

The verification has been performed with the 32-bit version of Uppaal 4.1.24 on a laptop with Intel i7-8665U 2.11 GHz CPU, running Ubuntu 22.04. The abstract models were generated using a script in node.js. The results are presented in Table 2. Each row lists the scalability factors (i.e., the number of voters and candidates), the size and verification time for the original model (so called “concrete model”), and the results for Abstractions 1, 2, and 3. “Memout” indicates that the verification process ran out of memory. The columns ‘ta’ and ‘tv’ stand for the abstract model generation and verification time, respectively. In all the completed cases, the verification of the abstract model was conclusive (i.e., the output was “true” for all the instances in Table 2).

\(^7\text{Implementation prototype and utilized models can be found at https://tinyurl.com/363pvpu5 and https://tinyurl.com/3eukkrkb.}\)
The results show significant gains. In particular, for the variant with \( NC = 3 \) candidates, our \( may \)-abstractions allowed to reduce the state space by orders of magnitude, and increase the main scalability factor by 3, i.e., to verify up to 9 instead of 6 voters.

## 7 Conclusions

In this paper, we present a correct-by-design method for model reductions that facilitate formal verification of MAS. Theoretically speaking, our reductions are agent-based may/must abstractions of the state space. Crucially, they transform the specification of the system at the level of agent graphs, without generating the global model. No less importantly, they are easy to use, come with a natural methodology, and require almost no technical knowledge from the user. All that the user needs to do is to select a subset of variables to be removed from the MAS graph representing the system. It is also possible to define mappings that merge information stored in local variables of an agent module.

We prove that the abstractions always generate a correct abstract MAS graph, i.e., one that provides a lower (resp. upper) bound for the truth values of formulae to be verified. Moreover, we demonstrate the effectiveness of the method on a case study involving the verification of a postal voting procedure using UPPAAL. As shown in the experiments, simple abstractions allow to verify state spaces larger by several orders of magnitude. Clearly, the efficiency of the method depends on the right selection of variables and the abstraction scope; ideally, that should be provided by a domain expert.

In the future, we want to combine variable abstraction with abstractions that transform locations in a MAS graph. Even more importantly, we plan to extend the methodology from branching-time properties to formal verification of strategic ability (Alur et al. 2002). We also note that the procedure is generic enough to be used in combination with other techniques, such as partial-order reduction (Jamroga et al. 2020a).
Acknowledgments

We thank Marius Belly-Le Guilloux, Damian Kurpiewski, Peter Y.A. Ryan, and Masoud Tabatabaei for comments and discussion. The work was supported by NCBR Poland and FNR Luxembourg under the PolLux/FNR-CORE projects STV (POLUX-VII/1/2019 and C18/IS/12685695/IS/STV/Ryan), SpaceVote (POLUX-XI/14/SpaceVote/2023 and C22/IS/17232062/SpaceVote) and PABLO (C21/IS/16326754/PABLO), as well as the CHIST-ERA grant CHIST-ERA-19-XAI-010 by NCN Poland (2020/02/Y/ST6/00064).

References


