PhD-FSTM-2024-024
The Faculty of Science, Technology and Medicine

DISSERTATION

Defence held on 17/04/2024 in Esch-sur-Alzette

to obtain the degree of

DOCTEUR DE L'UNIVERSITÉ DU LUXEMBOURG

EN INFORMATIQUE

by

Yan KIM
Born on 8 September 1994 in Karaganda (Kazakhstan)

FORMAL METHODS FOR ANALYSIS OF SECURE, RELIABLE, AND VERIFIABLE VOTING SCHEMES

Dissertation defence committee

Dr Marcus Völp, Chairman
Professor, Université du Luxembourg

Dr Wojciech Penczek, Vice Chairman
Professor, Institute of Computer Science of the Polish Academy of Sciences

Dr Wojciech Jamroga, dissertation supervisor
Research Scientist, Université du Luxembourg
Professor, Institute of Computer Science of the Polish Academy of Sciences

Dr Peter Y.A. Ryan
Professor, Université du Luxembourg

Dr Carsten Schürmann
Professor, IT University of Copenhagen

UNIVERSITY OF LUXEMBOURG

DOCTORAL THESIS

Formal Methods For Analysis Of Secure, Reliable, And Verifiable Voting Schemes

Author:
Yan Kim

Supervisor:
Prof. Dr. Wojciech Jamroga

Co-supervisor:
Prof. Dr. Peter Y.A. Ryan

Luxembourg, 2024

Abstract

Voting procedures are of utmost importance for society and are widely believed to be *the foundation* of democracy. They appear in different forms, in various scenarios, and often within certain legal framework and operational context; naturally, they encompass a broad range of interdisciplinary features and can be studied on different levels (e.g., voting scheme, stand-alone voting mechanism, implementation, etc.) under different perspective (e.g., legal, social, security, etc.). Recently proposed, solutions for electronic voting offer a great range of improvements and enhancements to speed, accuracy and overall efficiency, not to mention promises of integrity and transparency. Their design and implementation are highly challenging tasks, which should also be carefully planned to avoid shortcomings and thus not undermine the voters' trust in the technology itself. The formal analysis must consider both the technological side and the human/social context, in which the system is embedded.

This monograph will focus on the verification of socio-technical aspects of voting procedures using temporal model checkers.

Firstly, using the multi-agent formalism in our model we obtain a modular and highly scalable representation of the system, as well as more realistically capture interactions between human agents. The latter, unlike computer programs or machines, can often deviate from their prescribed protocol: either intentionally (with some goal in mind) or accidentally (due to mistake or lack of attention/understanding), and are extremely difficult to predict. In our case study, we attempt to accommodate (and often combine) ideas proposed in academic literature to consider human aspects in the model.

A more sophisticated model with more details naturally leads to challenges associated with computation complexity arising sooner. This is mainly caused due to the number of possible states of the system growing exponentially in the number of components and their details (e.g., processes and variables). This brings us to the second main contribution, which is the novel method for *practical model reduction* through variable-based abstraction. Furthermore, we establish the terminology, which combines features from both mathematical objects (e.g., concurrent-game structures) and practice-oriented ones (e.g., program graphs), in a way that preserves flexibility without abstracting much from the actual syntax of input models. We demonstrate how to employ it on examples of case studies and evaluate the provided benefits. Notably, with its help we were able to conduct verification of non-trivial configurations of a voting model even with a generic personal laptop. While, this is a speculation, the potential results on high-performance servers with more computational power/resources should be even more evident.

Lastly, we also show that the applicability of our method goes beyond that of voting schemes, but is general enough for other kinds of multi-agent systems.

Acknowledgements

First of all, I would like to thank my supervisors, Prof. Wojciech Jamroga and Prof. Peter. Y.A. Ryan, for granting me an exceptional opportunity to work on this research and for their immense help as well as unwavering support over the years it took me to complete this thesis. I would like to give special thanks to Wojciech for the kind mentorship, in both professional and personal aspects of life. In particular, I thank him for teaching me how to write academic papers, coaching me through my first presentations, as well as exceptional patience, especially when reading all the drafts or enduring my rehearsals. I also appreciate his tireless efforts to help me improve my communication skills.

I thank the members of my CET commission, Prof. Wojciech Jamroga, Prof. Peter. Y. A. Ryan and Prof. Wojciech Penczek, for counselling me throughout the PhD studies and providing expert feedback on my ideas. They also provided helpful remarks on the final structure of the thesis and pointed out some of my grammatical errors.

I am grateful to my former supervisors back at Nicolaus Copernicus University in Toruń, Prof. Piotr Dowbor and Dr. Marcin Piątkowski, for their guidance, encouragement and overwhelming belief in me, which often surpassed that of my own.

I want to thank my collaborators: Prof. Wojciech Jamroga, Prof. Peter Y.A. Ryan, Dr. Peter Roenne, Damian Kurpiewski, Dr. Łukasz Mikulski, Mateusz Kamiński, Witold Pazderski, Dr. Masoud Tabatabaei, Dr. David Mestel, and Marius Belly-Le Guilloux. There were a number of collaborations that did not result in co-authorship, but which certainly deserve recognition. In particular, Marius Belly-Le Guilloux read through the early draft of pseudocodes for variable-based abstraction, Masoud Tabatabaei gave valuable feedback on the earlier draft of the proposed formalism for our abstraction method, and Damian Kurpiewski and David Mestel provided helpful comments and suggestions for improving its presentation.

My gratitude also goes to Prof. Pierre Kelsen for entrusting me with the role of assistant for practical classes and for all the patience and support.

I thank all the members of the APSIA group and its alumni for providing a welcoming and supportive environment and many fascinating discussions, often over exotic sweets and coffee. I feel privileged to have been part of this group and to have had the opportunity to work with such interesting people. In particular, I want to thank my wise seniors, Peter Roenne, Afonso Delerue Arriaga and Marjan Skrobot, for uncountably many valuable hints and suggestions, and my former office-mates, Aditya Damodaran and Fatima El Orche, for many uplifting discussions and help in various matters.

I would also like to thank members of the TDCS research group from the Polish Academy of Science for the productive exchange of research ideas despite our geographical distance.

Last but not least, I would like to thank my whole family for their support and unconditional love: my dear parents Anastassiya and Vladimir, aunt Małgorzata and uncle Andrzej, and cousins Karol, Łukasz, Marek and Mateusz.

This work was supported by the National Centre for Research and Development, Poland (NCBR), and the Luxembourg National Research Fund (FNR), under the PolLux/FNR-CORE project STV (POLLUX-VII/1/2019 & C18/IS/12685695/IS/STV/Ryan).

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Voting is a mechanism that plays a critical role in many social processes and is often regarded as the *cornerstone* of democracy. It comes in various forms (conventional, electronic and hybrid) and features a plethora of application scenarios. Among the common application domains are: government elections and/or referenda, university (and other organizations) internal decision-making, corporate statutes, papal conclave, etc.

In recent years, there were numerous voting schemes proposed, which offer various improvements in almost every possible aspect of the process [Rya+09; Rya10; Riv06; Bel+13; Cul+15]. A notable example is e-voting [JCJ05; AM07; Adi08; RRI16; HR16; CGG19; CFL19], whence solutions can greatly increase the efficiency (e.g., reduce the managing cost, also in terms of time required before/after the actual voting), if implemented correctly are often more reliable, and mainly designed to be resistant to possible malicious behaviour of some participants. However, apart from new solutions, there are also novel threats and sophisticated attacks that should be considered. With e-voting in particular, the stakes are very high: not only does the voting system have to operate correctly, but also believed to do so. Examples of public discontent outbreaks — often in the form of protest or even riot — caused by the failure to provide credible or convincing enough results can easily be found all over the world. In the recent decade alone, we have witnessed the drastic magnitude of societal events during the US presidential elections 2016 and 2020, the Belarusian presidential election 2020. Altogether this brings up great challenges in the design, implementation, and analysis of voting protocols.

In this research, we will focus on the analysis of secure, reliable, and verifiable voting schemes, explore the application of automata-based techniques from formal methods and endeavor to combine certain concepts from adjacent disciplines, such as game theory and multi-agent systems. The goal is twofold: firstly, to provide stronger guarantees by conducting verification of voting systems, and secondly, to ensure

disambiguation of properties/requirements through their formal specification. The latter would further facilitate a better understanding of how the system works and how it should work, as well as help better recognize some of their internal differences and similarities.

Research Hypothesis

This thesis is about formal verification of voting procedures (including the socio-technical aspects) using temporal model checking. The research hypothesis is therefore stated as follows:

> Voting procedures, which are intrinsically versatile and typically notoriously complex systems from the start or eventually evolving into such, can be verified using state-of-the-art model checking tools to provide solid guarantees against well-defined system requirements.

1.1 Background

There might be no universal consensus on what the ultimate requirements of the voting system must be — depending on the scenario the set of requirements as well as their interpretation may be greatly varying. Nonetheless, we can still describe some commonly accepted ones and point to interesting further studies on that.

1.1.1 Privacy

One of the fundamental requirements for the voting system is that a voter's choice must be kept secret. Traditionally, this is broken down into the following:

(BP) Ballot privacy: a system must not reveal for whom a voter voted;

(RF) Receipt-freeness: a system must ensure that a voter cannot convince/prove to a third party how she voted;

(CR) Coercion-resistance: a system must ensure that a voter cannot cooperate with a coercer to prove how she voted.

Article 21(3) of the Universal Declaration of Human Rights [Uni48] demands the right for *secret vote*, which is often used interchangeably with (BP) but might also have a distinct interpretation in general.[1]

In [DLL12] Dreier et al. propose a modular family of privacy notions and identify the underlying (multidimensional) hierarchy of those. Furthermore, they demonstrate how it can be applied to some existing voting schemes, which also allows for a more systematic comparison of the latter. In particular, they show how proposed notions map to different properties of real-world protocols, and are possibly verified using an automated tool (such as ProVerif [BAF08; Bla+16], ProSwapper [KSR10]). Notably, the authors explicitly warn about the limitations associated with the possibilistic nature of an employed approach.

A comprehensive analysis of game-based computational notions for vote privacy, in particular, suitable for remote voting protocols, was discussed by Bernhard et al.

[1] Somewhat surprisingly, though, it is not uncommon in the literature to redefine or propose another interpretation of an already used term and typically without changing its name.

in [Ber+15]. The Additionally, they introduce new game-based definitions of privacy, called *BPRIV* and *strong consistency* that aim to address some limitations with existing ones and among others features that can account for possible tallying operations.

Another noteworthy study of proposed definitions of (RF) and (CR) was made in [TJR16], which focuses on the strategic aspects of that. More precisely, they overview existing (intuitive) definitions and present their interpretation/transcription of those using an alternating-time temporal logic **ATL** [Alu; Alu+98; AHK02]. This work demonstrates how coercion-related properties could be captured using the notion strategies (to coerce or defend), and also facilitates in better understanding of subtle differences behind them.

1.1.2 Verifiability

Verifiability aims to provide assurance that the outcome is accurate. Typically, one distinguishes two notions/kinds of it:

(IV) Individual Verifiability: a voter can verify that her vote was properly counted, and

(UV) Universal Verifiability[2]: anyone can verify that the result corresponds to the votes cast.

In the context of electronic voting, a commonly discussed verifiability technique is the end-to-end verifiability **(E2E-V)** [Ben+15; RST15; JMP13]. Traditionally, it requires the following three phases/tasks to be achieved/performed:

i. Cast-as-intended (CAI): a voter can check that her choice was accurately captured (e.g., on a paper ballot, or by digital means);
ii. Recorded-as-cast (RAC): a voter can check that her vote was received the same way it was cast;
iii. Tallied-as-recorded (TAC): a voter can check that her vote was correctly included in the tally.

Another interesting aspect of verifiability includes:

(ElV) Eligibility verifiability: anyone can verify that all votes in the results/tally originate/come from registered/eligible voters, and there is at most one vote per voter [KRS10].

Ideally, verifiability should render any malicious tampering of results detectable, while keeping the minimum amount of trust one needs to place (e.g., in a device, network infrastructure, officials, etc.).

Verifiability and privacy (or secrecy) are commonly recognized as two fundamental security attributes that a voting system should guarantee. Due to the almost conflicting nature of these requirements, designing a system that provides both can be a big challenge.

1.1.3 Other Important Properties

In verifiable e-voting schemes, vote privacy often relies on certain computational hardness assumptions[3] associated with employed cryptographic primitives. An

[2] First defined by Sako and Kilian in [SK95]

[3] That is a (widely believed) hypothesis that particular problem cannot be solved certain problems efficiently, i.e. in a reasonable amount of time (typically implied "polynomial time") and using currently available computational resources [Sti05].

interesting notion of privacy, called *everlasting privacy* [MN06], aims to ensure that voting secrecy can be preserved unconditionally anytime in the future. Achieving such without significant trade-offs can be notoriously difficult or even impossible, therefore a weaker/relaxed variant, called *practical everlasting privacy* [Ara+13], which restricts the power of adversary to perform certain operations on data, is often of strived for instead. A well-rounded overview of academic literature related to electronic voting protocols with everlasting privacy can be found in a recent SoK paper [Hai+23] by Haines et al.

There is also a whole family of properties related to *dispute resolution*, which investigates reports of system malfunctions. In particular, *accountability* ensures that "misbehaviour" can be traced back to the responsible party, which also provides a stronger incentive for participants to follow the protocol "honestly". Different levels of accountability, as well as their relation to verifiability, were studied by Küsters et al. in [KTV10b]. Their work also advocates the goal-centered definition of accountability that interprets "misbehaviour" in connection with the violation of a desired goal.

1.2 Examples

In this section, we provide a number of selected examples of proposed ways of implementing certain security properties for voting systems. Expectedly, as with an abundance of variants of requirements, we will cover only some mechanisms, mainly focusing on conceptual ideas rather than underlying technical components.

Cryptographic primitives

Since early suggestions of using cryptography for voting in [BT94], researchers invented a multitude of new methods and techniques for applying cryptography to voting schemes. Those can provide formal guarantees for integrity and present elegant solutions to problems (often previously considered impossible). Notable examples involve, but are not limited to: blind signatures, anonymizing mixes, public-key encryption, homomorphic encryption, zero-knowledge proofs and secure multi-party computation. We refer the interested reader to [HR16] for more details and in-depth discussion on applications.[4]

Benaloh challenge

Benaloh challenge [Ben06; Ben07] is a "cut-and-choose" technique that allows voters[5] to audit the encryption of their vote (or other form of made selection) and to check whether it was created/represented accurately. In particular, the vote creation device must commit to the encryption upon selection made by a voter. Next, a voter can decide whether to cast or to audit it. The latter provides a verifiable decryption, "spoils" the vote and prompts the voter to start over (possibly changing the selection made).[6] An audit can be repeated until the voter is convinced that a device behaves "honestly". A device, even if malicious, can never know in advance whether a certain vote will be audited, and thus cannot "cheat" without risk of being caught.

[4] Additional background information for readers unfamiliar with cryptography can be found in textbooks, e.g. [Sti05; KL07].

[5] Note, that original work [Ben06] does not restrict the usage to voters per se; a discussion of other scenarios goes beyond the scope of the present work.

[6] An audited encryption of the vote must never be submitted, as that would break the vote secrecy.

It is one of the most popular, if not the de facto, ways/mechanisms of implementing (CAI) in e-voting systems. It is generic enough and compatible with many application scenarios. Importantly, this mechanism, when implemented correctly, does not compromise the secrecy and receipt-freeness. Last but not least, it puts much strain on the voters, as, the auditing is first of all completely optional.[7]

Its mechanism was analysed under the game-theoretic model by Culnane and Teague in [CT16], whence one of the results was concluding the lack of "natural" rational strategy for the voter, which might call into question whether audits using Benaloh challenges are indeed sensible. This was recently addressed by Jamroga in a follow-up study [Jam23], where both technical and conceptual aspects of analysis as in [CT16] were examined more scrupulously; in consequence: it corrected some claims, pointed out certain important assumptions, which were previously implicit, and discussed application of rationality concepts beyond that of Nash equilibrium itself.

Fake material

For instance, in JCJ voting scheme [JCJ05] a voter is provided with fake credentials that can be used to deceive a potential coercer on how the vote was cast. Registered (encrypted) votes with credentials are publicly available on a bulletin board. Fake votes are later removed during the cleansing phase before tallying; this is preceded by the mixing of votes, which intends to eliminate the linkage to initially cast votes. In that way, a mechanism for coercion resistance can be implemented.[8]

Another interesting approach, which guarantees (RF), was proposed in [RRI16] and [ZRR20], where voters are secretly assigned with unique trackers. The pool of trackers is public, and votes are published in plaintext along with the encryption of the tracker on a World Bulleting Board. Some time after all votes were cast, the voter can either request a special alpha-term, which can be used to derive her tracker and verify it, or she may also request a fake term, which would point to the selected vote (e.g., demanded by a coercer). The scheme proposes a good level of transparency and an interesting mechanism for mitigating coercion attacks.

As can be seen from the above discussion, there is a multitude of interesting properties, and ways to define, interpret and further implement them. Overall, the context plays an important role here; such aspects are legal constraints, relevant threats and considered attacks, operational environment and many others. It possibly comes as no surprise that ways of performing a formal analysis of voting protocols come in great variety as well. A formal taxonomy of such might be a subject in itself and requires greater expertise in the field than presently available. Therefore, in the upcoming section, we will restrict ourselves to a brief description of a selected number of works, without attempting to establish a complete family of verification methods.

[7] It should be noted, however, that Benaloh challenge in principle does rely on voters to report if the device was caught in cheating during an audit.

[8] Although in [CGY22] it was later found that originally proposed variant [JCJ05] used a much weaker notion of (CR) and overlooked some potential exploits, the reported flaws were not related to the concept itself.

1.3 Model Checking

Model specification

The system components and their interactions can be modelled in various ways. Among the standard and widely used model representations describing behaviour of a system are:

- Transition System (TS) — a directed graph, where nodes represent the current state of the system and edges describe the transitions, state evolution/changes. Two of the most commonly used variants are: Kripke Structure (KS), where each state is associated with a subset of atomic properties it satisfies, and Labelled Transition System (LTS), where edges are associated with an action.[9] Furthermore, in case of 3-valued formalism, notions of Partial Kripke Structures (PKS), Modal Transition Systems (MTS) and Kripke Modal Transition Systems (KMTS) are often used instead.[10]
- Büchi Automaton (BA) — (non-deterministic) automaton accepting omega-regular languages. A variant, called generalized Büchi automaton is used in algorithms for verification of omega-regular properties.
- Program Graph (PG) — often used for modelling the data-dependent systems, usually represented in the form of a directed graph over the set of typed variables, where nodes represent control locations and edges as conditional transitions. Program graph could be interpreted as a transition system by taking the pairs of location and evaluation of variables for the states, effectively "unwrapping" the program graph. Usually, this provides a more compact view of the system than TS kinds or automata-based representations.
- Timed Automata (TA) — essentially a PG extended with a (finite) set of special real-valued variables called clocks; these allow for capturing the behaviour of the time-critical systems. [11]
- Markov Chain and Markov decision process (MDP) — both enrich TS with probabilistic choices; intuitively, the Markov chain replaces the nondeterministic choice of the state's successors with probability distribution, whereas MDP supports the co-existence of both nondeterministic and probabilistic choices, which allows for an adequate model of randomized distributed systems.

Dealing with SSE

One of the biggest challenges in applying model checkers for verification of real-world systems remains a phenomenon known as a state-space explosion (SSE). This occurs due to the number of states growing exponentially in the number of system components (e.g., sub-processes) and their details (e.g., variables). In attempt to mitigate this problem several techniques were proposed:

- Abstraction — a technique based on the idea of eliminating details of the system, that are irrelevant for the verification of a property, so that a resulting smaller model is used. In many cases analysis of an abstracted model will be sufficient to decide the problem for the original one.

[9] In many contexts KS and LTS can be used interchangeably, as their distinction is mainly technical, allowing results to be formulated using either formalism.

[10] We omit the details, as the topic goes beyond the scope of this work. An interested reader can refer for more details to [God14].

[11] Here, we refer to the (more intuitive) definition from [Hen+94; BY03; BK08] instead of that originally introduced in [Alu99].

- Under-/over-approximation (or simulation) — two commonly used approaches that rely on a partitioning of the state space, effectively reducing the size of a system. Such reduction is done in a way that guarantees preservation of certain fragment of CTL* formulae (existential for under-approximation, universal for over-approximation).
- Partial order reduction — an on-the-fly method of state space reduction that has been defined for linear and branching temporal logics, and more recently also applied for the verification of strategic ability using ATL. Rather than unfold the global system graph based on all available transitions, the idea is to restrict to a provably sufficient subset (commonly referred to as the ample set, stubborn set or persistent set) in each global state. As the exact computation of this subset is costly, heuristics have been defined. The obtained reduced model satisfies temporal and temporal-strategic formulae iff they are satisfied in the full model (which, thanks to POR, is never generated).
- Counter-example guided refinement (CEGAR) — automatic iterative-refinement methodology defined for universal fragment of CTL* specification. It computes an over-approximation of a concrete system and then checks specification against it. If a specification is not satisfied in the abstract system and the counterexample run is not present in the original system, the abstraction is refined in way, preventing such erroneous behaviour from happening and the verification step is repeated.
- Symbolic verification using Binary Decision Diagrams — in this paradigm, rather than using explicit, graph-based data structures for automata that comprise the verified model, the transition relation and sets of states are represented with binary decision diagrams (BDDs). This representation is sufficient for implementing model checking algorithms (e.g., fixpoint-based) for a range of temporal logics. Because it is much more concise than explicit structures, it enables the practical verification of significantly larger models, including real-world systems and protocols used in the industry.
- Bounded model checking using satisfiability solving — an alternative approach to symbolic model checking that uses boolean satisfiability (SAT) procedures. The idea is to construct a formula that is satisfiable iff there exists a counterexample of a specific length (increased, up to a given bound, in subsequent iterations). It uses less space than BDD-based approaches and quickly finds counterexamples of minimum length due to the depth-first nature of SAT-solving algorithms.

Some tools

Applying model checking is typically split into three parts: establishing the model of possible system behaviour, formalizing the specification of requirements, and executing the verification. A concise overview of these processes with respect to selected verification tools is given below:

- MCMAS — a state-of-the-art, OBDD-based symbolic model checker. The description of multi-agent systems (MAS) is given by means of ISPL (Interpreted Systems Programming Language) format. This specification is agent-based and each agent is described by providing a set of variables, available actions, agent protocol and evolution functions. Among others, MCMAS supports CTL and ATL formulae for specifying requirements for the system. It also supports basic fairness conditions and allows to generate counterexamples. The tool can be used from the shell, but Eclipse-based graphical interface is also provided.
- SPIN — a software verification tool and full LTL model checker. The system is described in PROMELA (process metalanguage), which supports embedding C

code as part of the model specification. Apart from LTL formulae, the requirements can be also specified as system/process invariants, Büchi automata, and general omega-regular properties. Both rendezvous and buffered communication are supported, and SPIN is one of the few verification tools that allows for a dynamically changing number of processes. Furthermore, it supports partial order reduction, which coupled with on-the-fly verification allows for efficient handling of large systems.

- TAMARIN prover — a tool aimed at security protocol verification. It is not a model checker but can be used as a helper tool for symbolic modelling and analysis of protocols heavily based on cryptography, such as most of the e-voting protocols. Its specification language is based on multiset rewriting systems. The systems are analysed with respect to (temporal) first-order properties. As other theorem provers, TAMARIN inherits greater expressive power but requires high proficiency to be used effectively.
- UPPAAL — provides an environment for modelling and verification of a real-time system, represented by a network of extended timed automata. System components (and their templates) are described using C/C++ style language and stored either as XML or as XTA files (+ optional UGI file for the layout). Requirements specification support is limited to a subset of CTL* formulae (excluding next and until modal operators, and nesting of path quantifiers). The verification process comes to on-the-fly exploration of the global model with an option to select which traversal algorithm will be used (BFS, DFS, randomized DFS). Verification results with one of the following messages: "property is satisfied", "property is not satisfied" with violating run, and "out of memory".
- PRISM — at the moment the most well-known probabilistic model checker, it supports a wide range of probabilistic models, including discrete Markov chains, continuous-time Markov chains, Markov decision processes, probabilistic automata and probabilistic timed automata, which combine concepts from TA and MDP. Its property specification language incorporates PCTL, CSL, LTL and PCTL* temporal logics as well as extensions for quantitative specifications and costs/rewards. It provides multiple model checking engines (BDD-based symbolic, explicit-state) and implements a variety of verification techniques, such as symmetry reduction and quantitative abstraction refinement.
- STV (StraTegic Verifier) — an experimental tool for verification of strategic abilities under imperfect information and strategy synthesis guaranteeing a given temporal goal. The model is specified as an asynchronous multi-agent system (in other words, a network of local automata). Properties are expressed using alternating-time temporal logic ATL and encoded with the model in the input file. The tool performs verification using explicit-state model checking with support for automated partial order reduction and A-bisimulation (bisimulation checking if a pair of models is bisimilar for a given coalition) for provided bisimilation relation.

1.4 Contributions

The contributions of this thesis fall under two main strands. The first concerns the proposed variable abstraction method, which is of both technical and innovative nature. It is designed to operate on the compact (often modular) representation of the system, can process the input and output independently of the chosen model checking environment, and guarantees the generation of the correct-by-design abstract models. In contrast to many existing (and more general) state abstraction techniques, this

solution does not require generation of complete concrete state space[12] beforehand, which, in many cases, is exactly the bottleneck that we want to avoid. Hence, the novelty comes not from *what* is being done — the principles of the state abstraction are more than 30 years old — but in *how* it is specified. For that purpose, it was necessary to work out an appropriate formalization framework, including the right terminology, which is flexible enough yet not too abstract, so that it is ready to be applied in practice as is. The abstraction is specified at a high level, supports various operations on variables (e.g., removing them, merging them into a new variable, restricting their domain, etc.) and allows to restrict the effect to a particular fragment of the model only (specified by a subset of locations). Overall this provides an intuitive yet powerful syntax for specifying an abstraction. Given abstraction parameters the generation process can be fully automated and does not require the user to understand its theoretical underpinnings, as produced abstract models are guaranteed to be sound. The proposed algorithm is supported by correctness proofs and complexity analysis.

Furthermore, we provide an implementation of the abstraction method for UPPAAL model checker, which we utilized in the series of experiments with different models. In particular, we demonstrate how it can be used on example of realistic case studies and evaluate its effectiveness.

Consequently, this brings us to the second contribution cluster related to the modelling and verification of multi-agent systems from the voting protocols. We used model checking for verification of Prêt à Voter, Polish Postal Voting — first the simplified one and then a more mature/sophisticated one — and Social Explainable AI protocol. The proposed models have a modular structure, are parameterized by configuration and are highly scalable in general. Having a flexible and transparent model specification is useful in many ways, not only does it help with preliminary validation in the early stages of modelling, and further maintenance in the later stages,[13] but it also allows for possible refinement of the model in the future (including the addition of new agents), encourages the reuse of existing module specifications, and improves the overall efficiency of the modelling process, especially when multiple variants of the system are to be examined or compared. Furthermore, the use of logical formalism to express the properties of the modelled system provides an unambiguous and reasonably intuitive interpretation.

While the study did not result with an ultimate prescription (or complete/matured methodology) and neither verification discovered new kinds of attacks, it is an important step towards that. Indeed, capturing a broad spectrum of possible behaviours of the system and its components (e.g., system communication with the environment or agent-to-agent interactions) within a model, let alone verifying such a complex system, might be infeasible in its entirety with current or foreseeable in the near future tools. However, even when done on a smaller scale, the analysis will allow detecting multitude of bugs (or other forms of undesired states of the system) and in their absence increase the confidence in the correctness of the system.[14]

[12] This requirement can also be stated implicitly; for example when underlying equivalence relation (over the concrete states) is given in parametric form.

[13] Both of these features are intrinsically crucial for ensuring that what has been modelled genuinely corresponds to what we think/believe has been modelled. Recall that "any verification using model-based technique is only as good as the model of the system"[BK08]

[14] There might be an argument that possibly false-positive (or erroneous) confidence in the system is of a great risk on itself, however it must be stressed that, as with almost any kind of analysis, the derived results are inherently bound to the assumptions; here, the outcome, when no violations of security requirements were found, relates to level of abstraction from real system to its model and considered threats.

Moreover, we describe various modelling practices, including technical optimizations and tricks, which can greatly facilitate the analysis. In particular, we shed light on certain phases of capturing systems and their requirements through models and formal properties, which are often omitted in the academic literature due to lack of scientific value or novelty, but which can be helpful to readers with less expertise. For example: dealing with state space explosion, generating input models from a more expressive specification (e.g., via template engine), and managing and benchmarking families of models.

Lastly, this thesis provides a systematic overview of state-abstraction techniques, approaches to modelling human behaviour (including their possible mistakes or other forms of deviating from the expected, "honest" behaviour) and existing studies of socio-technical aspects of voting schemes. No less importantly, our work explores a different modelling paradigm that offers a more natural representation of the system in terms of agent-based semantics and an approach that was strongly driven by practical concerns.

1.4.1 Publications

In the course of doctoral studies I had three publications accepted in A*-ranked[15] and two in B-ranked[16] conferences/journals. This thesis mainly builds on material from the following papers:

- *"Towards Model Checking of Voting Protocols in Uppaal"*, on which Chapter 3 was based. The co-authors of this paper were: Wojciech Jamroga, Damian Kurpiewski, and Peter Y. A. Ryan. It was published in Proceedings of the 5th International Joint Conference on Electronic Voting, E-Vote-ID 2020, Springer LNCS.
- *"Verification of the Socio-Technical Aspects of Voting: The Case of the Polish Postal Vote 2020"*, on which Chapter 6 was based. The co-authors of this paper were: Wojciech Jamroga and Peter Y. A. Ryan. It will appear in Proceedings of the 12th International Workshop on Socio-Technical Aspects in Security, STAST 2022, Springer LNCS.
- *"Practical Abstraction for Model Checking of Multi-Agent Systems"*, on which Chapter 4 was based. The co-author of this paper was Wojciech Jamroga. It was published in Proceedings of the 20th International Conference on Principles of Knowledge Representation and Reasoning, KR 2023 (A* rank in CORE).
- *"Practical Model Reductions for Verification of Multi-Agent Systems"*, on which Chapter 5 was based. The co-author of this paper was Wojciech Jamroga. It was published in Proceedings of the 32nd International Joint Conference on Artificial Intelligence, IJCAI 2023 (A* rank in CORE).
- *"Scalable Verification of Social Explainable AI by Variable Abstraction"*, on which Chapter 8 was based. The co-authors of this paper were: Wojciech Jamroga and Damian Kurpiewski. It will appear in Proceedings of the 16th International Conference on Agents and Artificial Intelligence, ICAART 2024 (B rank in CORE), SCITEPRESS.

The publications — that were written in the course of the doctoral studies but were not included in the scope of discussion for this thesis — are listed below:

[15] A* is the highest CORE ranking assigned to the "flagship conference/journal, a leading venue in the discipline area" [Com24].

[16] B is the CORE ranking assigned to a "very good conference/journal, and well regarded in a discipline area" [Com24].

- *"Computational Classification of Tubular Algebras"*, which was co-authored with Piotr Dowbor. It was published in Fundamenta Informaticae vol. 177 (B rank in CORE), IOS Press.
- *"STV+Reductions: Towards Practical Verification of Strategic Ability Using Model Reductions"*, which was co-authored with: Wojciech Jamroga, Damian Kurpiewski and Witold Pazderski. It was published in Proceedings of the 20th International Conference on Autonomous Agents and Multiagent Systems, AAMAS 2021 (A* rank in CORE), IFAAMAS.
- *"You Shall not Abstain! A Formal Study of Forced Participation"*, which was co-authored with: Wojciech Jamroga, Peter Y.A. Ryan and Peter Roenne. It will appear in Proceedings of the 9th Workshop on Advances in Secure Electronic Voting, Voting 2024.

1.4.2 Presentations and Talks

Below are the conference presentations and seminar talks, in which I had the honour to speak about my research, listed in chronological order:

- *"Pret-a-Voter in Uppaal"* Internal Seminar of Secure, Usable and Robust Cryptographic Voting Systems (SURCVS) Project, 04/09/2020, Online;
- *"Towards Model Checking of Voting Protocols in Uppaal"*, Applied Security and Information Assurance (APSIA) Research Seminar, 24/09/2020, Luxembourg;
- *"Towards Model Checking of Voting Protocols in Uppaal"*, Main Programme of the 5th International Joint Conference on Electronic Voting, E-Vote-ID 2020, 09/10/2020, Online;
- *"Towards Model Checking of Voting Protocols in Uppaal"*, Theory of Distributed and Computing Systems (TDCS) Group Seminar, 01/10/2020, Online;
- *"Towards Model Checking of Voting Protocols in Uppaal"*, PhD Colloquium of the 20th International School on Foundations of Security Analysis and Design, FOSAD 2021, 02/09/2021, Bertinoro, Italy;
- *"Formal Verification of Multi-Agent Systems"*, Interdisciplinary Centre for Security, Reliability and Trust (SnT) Partnership Day 2021, 18/11/2021, European Convention Center, Luxembourg,
- *"Abstraction based on variable removal"*, Applied Security and Information Assurance (APSIA) Research Seminar, 13/12/2021, Luxembourg;
- *"Verification of the Socio-Technical Aspects of Voting: The Case of the Polish Postal Vote 2020"*, 12th International Workshop on Socio-Technical Aspects in Security, STAST 2022, 29/09/2022, Copenhagen, Denmark;
- *"Abstraction for Multi-Agent Systems with Uppaal"*, Theory of Distributed and Computing Systems (TDCS) Group Seminar, 09/02/2023, Online;
- *"Verification of the Socio-Technical Aspects of Voting: The Case of the Polish Postal Vote 2020"*, Formal Languages and Concurrency (FOLCO) Group Seminar, 14/03/2023, Toruń, Poland;
- *"Practical Abstraction for Model Checking of Multi-Agent Systems"*, Main Programme of the 20th International Conference on Principles of Knowledge Representation and Reasoning, KR 2023, 9/09/2023, Rhodes, Greece;
- *"Practical Abstraction for Model Checking of Multi-agent Systems"*, Internal Workshop of Probabilistic Verification Of Complex Heterogeneous Systems: From Ballots To Ballistics (SpaceVote) Project, 22/09/2023, Gdańsk, Poland;
- *"Verification and Modelling of Polish Postal Voting"*, PhD Colloquium of the 8th International Joint Conference on Electronic Voting, E-Vote-ID 2023, 03/10/2023, Luxembourg;

1.5 Structure of the Thesis

This monograph is self-contained. It is structured in 9 chapters. The chapter dependencies are illustrated in Fig. 1.1, where dashed lines denote that the abstraction tool from Chapter 5 was used in some experimental cases of Chapters 6 and 8.

The main chapters (3–8) are organized as follows: at the beginning, we briefly introduce the main topic of the chapter and then provide the content with necessary details. Each chapter includes a separate discussion of related work, a conclusion and directions for future work.

The outline of each individual chapter is given below:

Chapter 1 provides a gentle introduction to the subject of this work, it establishes the motivation and gives a brief overview of related literature.

Chapter 2 introduces the reader to the model checking primitives and basic notions necessary for further discussion.

Chapter 3 demonstrates how model checking tools like UPPAAL can be applied for the modelling and verification of voting procedures on the example of the Prêt à Voter scheme. It describes: modelling approach, encodings of some important voting properties and examination of Pfitzmann's attack on the randomized mix-nets that might break the privacy, and discusses certain optimizations aimed to enhance scalability and to keep the model manageable. Furthermore, we show how one does a model checking of temporal-epistemic properties via technical reconstruction of the input model and a purely temporal formula. This was the first step towards analysis of voting schemes using temporal model checkers, which showed that conceptually the results were promising, but also that computation complexity is a huge obstacle in practice.

Chapter 4 discusses the topic of practical model reduction for mitigation of the state space explosion. We overview a number of existing reduction techniques together with the limitations they impose. To cope with the latter, we proposed a conceptually simple method of variable-based state-abstraction. It is generic enough to be used/compatible with other techniques (e.g., partial-order reductions, sweepline) and constructed model reductions are correct-by-design. No less importantly, the presented method is pretty user-friendly: first-of-all, it can be used with minimal expertise[17], the user is not even required to understand how it works; secondly, it naturally operates on compact system representations, which not only aid in more efficient computation but are also already familiar to the user. Of course, to be used efficiently, in the sense of obtaining a significantly reduced model and conclusive result, a "good" selection of abstraction parameters must be made.[18] Another important contribution of this work is the establishment of a formal notation that preserves the flexibility of purely mathematical objects, albeit without diverging too far from the constructs that are actually used in practice.

Chapter 5 revisits the topic of practical abstraction and reports an experimental implementation of the aforementioned method. It gives an overview of the

[17] Both technical, as understanding the internal operations of the tool, as well as mathematical backbone standing behind the theoretical guarantee.

[18] This would usually rely on guidance from domain experts, but it might as well be done through an educated guess (recall that abstractions are generated in an automated way and are correct-by-design).

tool's architecture, application scenarios, and evaluation on examples of two benchmarks that were performed on the models of postal voting and gossip learning for social AI. The tool was designed with portability in mind so that using it does not inject any backwards dependability. Abstract models are available on the output and come in the same specification format as the input model; those can be further loaded and normally used with UPPAAL.

Chapter 6 presents a model of Polish Postal Voting protocol, with a focus on capturing socio-technical aspects and practical verification. The case study involves a family of parameterized models, including those with corrupted officials and fallible voters. It also describes an algorithmic approach for the translation of requirements from intuitive description to logical formula, which will be queried in the tool. In particular, it proposes interesting approximations allowing to overcome certain limitations caused by a lack of computational resources or narrowly supported specification language. Furthermore, the presented case study demonstrates the benefits of our abstraction method for analysis of a model of the real-world voting procedure.

Chapter 7 follows up the ideas presented in the previous chapter and further develops a model of Polish Postal Voting protocol and captures a greater variety of socio-technical interactions. In particular, it proposes a hierarchical modelling approach, which allows organising a rather compound system specification in a clear and easy-to-maintain manner. Furthermore, it also enhances the verification workflow and suggests, how a family of parameterized models with multi-dimensional configurations can be studied and discusses certain implications, which allow propagating the results from one onto many configurations, thus improving both efficiency and understanding of the system's internal dependencies.

Chapter 8 presents a case study of a family of SAI models. This attests that the applicability of our proposed abstraction method goes beyond voting procedures, and proves it to be beneficial also in the verification of other kinds of multi-agent systems. The case study covers 27 variants of SAI models, each having a number of its own configurations. Additionally, it proposed a clever modelling trick that allows to use impose standard interpretations of AF and EG formulae.

Chapter 9 provides a summary, concluding remarks and suggests possible directions for future research.

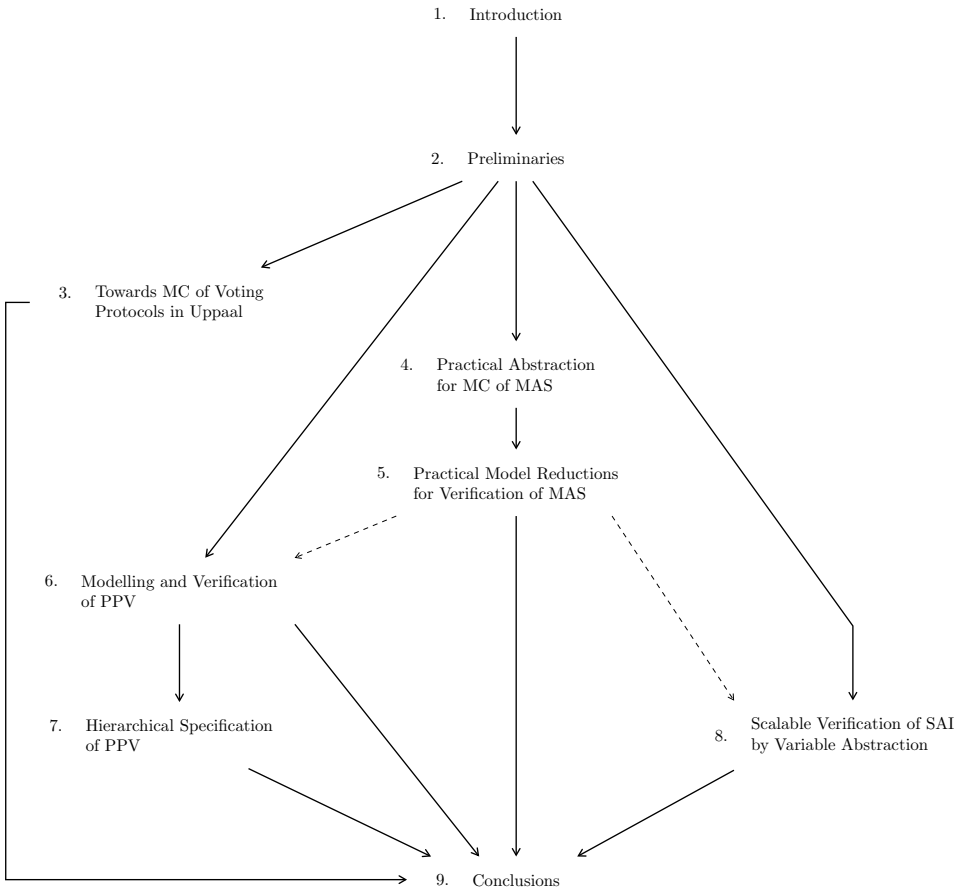


FIGURE 1.1: Chapter dependencies.

Chapter 2

Preliminaries

In this chapter, we recall from the literature the standard class of representations used for concurrent systems and some other model checking primitives, that will be necessary in further discussions.

Definition 2.1 (Model)**.** *A **model** is a tuple $M = (St, I, \longrightarrow, AP, L)$, where:*

- *St is a set of states,*
- *$I \subseteq St$ is a non-empty set of initial states,*
- *$\longrightarrow \subseteq St \times St$ is a transition relation,*
- *AP is a set of atomic propositions,*
- *$L : St \to 2^{AP}$ is a labelling function.*

We assume $\longrightarrow$ to be serial, i.e., there is at least one outgoing transition at every state, and St and AP to be finite.

Definition 2.2 (Run)**.** *A **run** of model M is a sequence of states $\pi = s_0 s_1 \ldots$, such that $s_i \in St$ and $s_i \longrightarrow s_{i+1}$ for every $i \geq 0$.*

A length of a run is denoted by $len(\pi)$, such that $len(\pi) = n$, if run is finite (i.e. $\pi = s_0 s_1 \ldots s_n$), and $len(\pi) = \infty$ otherwise. By $\pi[k]$ and $\pi[i, j]$ we denote the k-th state[1] of π and the fragment of π from index i to j respectively.

The set of all runs in M is denoted by $Runs(M)$, and the set of runs of certain length $t \in \mathbb{N}^+ \cup \{\infty\}$ is denoted by $Runs^t$.

Definition 2.3. *A **path** of model M is a **maximal** run, that is either infinite or ends in a state with no outgoing transitions (also called **terminal** state). A path that starts in initial state $s_0 \in I$ is called **initial** path.*

The set of all paths in M and all paths starting from state $s \in St$ are denoted by $Paths(M)$ and $Paths(s)$ respectively.

Unless stated otherwise, we shall also assume that St of a model M only contains states that are *reachable* from I, namely those $s \in St$ for which there exists a run $\pi = s_0 \ldots s_n s$, where $s_0 \in I$ and $s_i \in St$ for $i = 0 \ldots n$.

2.1 Temporal Logic

Temporal logic allows us to reason about the execution paths of the system (as sequences of transitions between its states). In this work, we will focus on a powerful branching-time logic called Computation Tree Logic **CTL** and its extension called **CTL*** [Eme90; CES86; EH86], which are widely used in the verification of reactive systems.

[1] An indexing of such states starts from 0.

Definition 2.4 (Syntax of **CTL**). *In* **CTL** *there are two types of formulas:* ***state formulae*** φ *and* ***path formulae*** ψ*. Their syntax over a set of atomic propositions AP is given by the following grammar:*

$$\begin{array}{lcl} \varphi & ::= & \mathsf{true} \mid p \mid \neg\varphi \mid \varphi \vee \varphi \mid \mathsf{A}\psi \mid \mathsf{E}\psi \\ \psi & ::= & \mathrm{X}\varphi \mid \varphi \,\mathrm{U}\, \varphi \end{array}$$

where $p \in AP$ is an atomic proposition, X *and* U *stand respectively for "next" and "until".*

The clauses for Boolean connectives are standard, and additional temporal operators "sometime" and "always" can be derived as follows:

$$\begin{array}{lcl} \textit{(sometime)} \quad \mathrm{F}\varphi & \equiv & \mathsf{true}\,\mathrm{U}\,\varphi \\ \textit{(always)} \quad \mathrm{G}\varphi & \equiv & \neg\mathrm{F}\neg\varphi \end{array}$$

Definition 2.5 (Semantics of **CTL**). *Let $M = (St, I, \to, AP, L)$ and $p \in AP$. Satisfaction relation $\models$ of a* **CTL** *state formula φ is given with respect to state $s \in St$, and of a* **CTL** *path formula ψ w.r.t. path $\pi \in Paths(M)$. It is inductively defined by:*

$$\begin{array}{ll} M, s \models p & \textit{iff } p \in L(s) \\ M, s \models \neg\varphi & \textit{iff } M, s \not\models \varphi \\ M, s \models \varphi_1 \vee \varphi_2 & \textit{iff } M, s \models \varphi_1 \textit{ or } M, s \models \varphi_2 \\ M, s \models \mathsf{A}\psi & \textit{iff } M, \pi \models \psi, \textit{for all } \pi \in Paths(s) \\ M, s \models \mathsf{E}\psi & \textit{iff } M, \pi \models \psi, \textit{for some } \pi \in Paths(s) \\ \\ M, \pi \models \mathrm{X}\psi & \textit{iff } M, \pi[1, \infty] \models \psi \\ M, \pi \models \psi_1 \,\mathrm{U}\, \psi_2 & \textit{iff } \exists j \geq 0.\, (M, \pi[j, \infty] \models \psi_2, \textit{and} \\ & \qquad \forall 0 \leq i < j.\, M, \pi[i, \infty] \models \psi_1) \end{array}$$

The model M is said to satisfy **CTL** *(state) formula φ (written $M \models \varphi$) iff $M, s_0 \models \varphi$, for all initial states $s_0 \in I$.*

The **CTL*** extends the (vanilla) **CTL** by allowing path quantifiers to be arbitrary nested with temporal ones[2], which also provides greater expressive power.

Definition 2.6 (Syntax of **CTL***). *The syntax of* **CTL*** ***state formulae*** φ *and* ***path formulae*** ψ *over a set of atomic propositions AP is given by the following grammar:*

$$\begin{array}{lcl} \varphi & ::= & \mathsf{true} \mid p \mid \neg\varphi \mid \varphi \vee \varphi \mid \mathsf{A}\psi \\ \psi & ::= & \varphi \mid \neg\psi \mid \psi \vee \psi \mid \mathrm{X}\psi \mid \psi \,\mathrm{U}\, \psi \end{array}$$

where $p \in AP$ is an atomic proposition, X *and* U *stand respectively for "next" and "until".*

The clauses for Boolean connectives are standard, and additional temporal operators "sometime" and "always" can be derived as follows:

$$\begin{array}{lcl} \textit{(sometime)} \quad \mathrm{F}\varphi & \equiv & \mathsf{true}\,\mathrm{U}\,\varphi \\ \textit{(always)} \quad \mathrm{G}\varphi & \equiv & \neg\mathrm{F}\neg\varphi \end{array}$$

Furthermore, in **CTL*** *the existential path quantifier can be defined as:*

$$\mathsf{E}\psi \quad \equiv \quad \neg\mathsf{A}\neg\psi$$

[2]Notice that in **CTL** every occurrence of a temporal operator is always immediately preceded with a path quantifier.

Definition 2.7 (Semantics of **CTL***)**.** *Let $M = (St, I, \to, AP, L)$, $s \in St$ and $\pi \in Paths(M)$. Satisfaction relation $\models$ for* **CTL*** *state formula φ and path formula ψ is inductively defined by:*

$$
\begin{array}{ll}
M, s \models p & \text{iff } p \in L(s) \\
M, s \models \neg\varphi & \text{iff } M, s \not\models \varphi \\
M, s \models \varphi_1 \vee \varphi_2 & \text{iff } M, s \models \varphi_1 \text{ or } M, s \models \varphi_2 \\
M, s \models \mathsf{A}\psi & \text{iff } M, \pi \models \psi, \text{ for all } \pi \in Paths(s) \\
\\
M, \pi \models \varphi & \text{iff } M, \pi[0] \models \varphi \\
M, \pi \models \neg\psi & \text{iff } M, \pi \not\models \psi \\
M, \pi \models \psi_1 \vee \psi_2 & \text{iff } M, \pi \models \psi_1 \text{ or } M, \pi \models \psi_2 \\
M, \pi \models \mathsf{X}\psi & \text{iff } M, \pi[1, \infty] \models \psi \\
M, \pi \models \psi_1 \,\mathsf{U}\, \psi_2 & \text{iff } \exists j \geq 0.\, (M, \pi[j, \infty] \models \psi_2, \text{ and} \\
 & \qquad \forall 0 \leq i < j.\, M, \pi[i, \infty] \models \psi_1)
\end{array}
$$

The model M is said to satisfy **CTL*** *(state) formula φ (written $M \models \varphi$) iff $M, s_0 \models \varphi$, for all initial states $s_0 \in I$.*

Often, techniques and methods dealing with state space explosion operate on more restricted variants of logic. In particular, the universal fragment of the branching-time logic **CTL*** called **ACTL*** (resp. universal fragment of **CTL** called **ACTL**) allows only universal path quantification. We formally introduce its syntax and semantics below.

Definition 2.8 (Syntax of **ACTL**)**.** *The syntax for* **ACTL** *over a set of atomic propositions AP is given by the following grammar:*

$$
\begin{array}{rcl}
\varphi & ::= & \mathsf{true} \mid \mathsf{false} \mid p \mid \neg p \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \\
 & & \mathsf{AX}\varphi \mid \mathsf{A}\varphi \,\mathsf{U}\, \varphi \mid \mathsf{A}\varphi \,\mathsf{R}\, \varphi
\end{array}
$$

where $p \in AP$ is an atomic proposition, and $\mathsf{X}, \mathsf{U}, \mathsf{R}$ *stand respectively for "next", "until", and "release".*

Formulas of **ACTL** *are required to be in* ***positive normal form****, i.e. where negations can be applied only to atomic propositions.*[3] *For this reason, both: conjunction and disjunction Boolean connectives, as well as until and release modal operators are given here explicitly. Additional temporal operators "sometime" and "always" can be derived as follows:*

$$
\begin{array}{lrcl}
\textit{(sometime)} & \mathsf{F}\varphi & \equiv & \mathsf{true} \,\mathsf{U}\, \varphi \\
\textit{(always)} & \mathsf{G}\varphi & \equiv & \varphi \,\mathsf{R}\, \mathsf{false}
\end{array}
$$

Definition 2.9 (Semantics of **ACTL**)**.** *The semantics of* **ACTL** *is given with respect to a state s in a model M is inductively defined by:*

[3] Notice that otherwise the existential path quantifier would be implicitly introduced.

$M, s \models p$ *iff* p *is on the list of labels* $L(s)$*;*
$M, s \models \neg p$ *iff* p *is not on the list of labels* $L(s)$*;*
$M, s \models \mathsf{A}\mathsf{X}\varphi$ *iff, for each path* $\lambda \in Paths(s)$*, we have* $M, \lambda[1] \models \varphi$*;*
$M, s \models \mathsf{A}\varphi_1 \,\mathsf{U}\, \varphi_2$ *iff, for each* $\lambda \in Paths(s)$*, there is* $i \geq 0$ *with* $M, \lambda[i] \models \varphi_2$*, and for each* $0 \leq j < i$ *it holds that* $M, \lambda[j] \models \varphi_1$*;*
$M, s \models \mathsf{A}\varphi_1 \,\mathsf{R}\, \varphi_2$ *iff, for each* $\lambda \in Paths(s)$ *and* $i \geq 0$*, we have* $G, \lambda[i] \models \varphi_2$*, or there is* $j \leq i$ *such that* $G, \lambda[i] \models \varphi_1$*.*

The model M *satisfies* **ACTL** *formula* φ *(written* $M \models \varphi$*) iff* $M, s_0 \models \varphi$*, for all initial states* $s_0 \in I$*.*

Definition 2.10 (Syntax of **ACTL***)**.** *The syntax for* **ACTL*** *over a set of atomic propositions* AP *is formally given by:*

$$\begin{array}{lcl} \varphi & ::= & \mathsf{true} \mid \mathsf{false} \mid p \mid \neg p \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \mathsf{A}\psi \\ \psi & ::= & \varphi \mid \psi \wedge \psi \mid \psi \vee \psi \mid \mathsf{X}\psi \mid \psi \,\mathsf{U}\, \psi \mid \psi \,\mathsf{R}\, \psi \end{array}$$

where $p \in AP$*, and* $\mathsf{X}, \mathsf{U}, \mathsf{R}$ *stand for "next", "until" and "release" respectively. Additional temporal operators "sometime" and "always" can be derived in the same way as for* **ACTL***. Just as in* **CTL****, formulae* φ *are called state formulae, and* ψ *are called path formulae.*

Definition 2.11 (Semantics of **ACTL***)**.** *Let* $M = (St, I, \rightarrow, AP, L)$*,* $p \in AP$*. Satisfaction relation* $\models$ *of a state formula* φ *is given with respect to state* $s \in St$*, and of a path formula* ψ *w.r.t. path* $\pi \in Paths(M)$*. It is inductively defined by:*

$M, s \models p$ *iff* $p \in L(s)$
$M, s \models \neg p$ *iff* $p \notin L(s)$
$M, s \models \varphi_1 \wedge \varphi_2$ *iff* $M, s \models \varphi_1$ *and* $M, s \models \varphi_2$
$M, s \models \varphi_1 \vee \varphi_2$ *iff* $M, s \models \varphi_1$ *or* $M, s \models \varphi_2$
$M, s \models \mathsf{A}\psi$ *iff* $M, \pi \models \psi$*, for all* $\pi \in Paths(s)$

$M, \pi \models \varphi$ *iff* $M, \pi[0] \models \varphi$
$M, \pi \models \mathsf{X}\psi$ *iff* $M, \pi[1, \infty] \models \psi$
$M, \pi \models \psi_1 \,\mathsf{U}\, \psi_2$ *iff* $\exists j \geq 0.\, (M, \pi[j, \infty] \models \psi_2 \;\wedge\; \forall 0 \leq i < j.\, M, \pi[i, \infty] \models \psi_1)$
$M, \pi \models \psi_1 \,\mathsf{R}\, \psi_2$ *iff* $\forall j \geq 0.\, (M, \pi[j, \infty]) \models \psi_2$ *or* $\exists j \geq 0.\, (M, \pi[j, \infty] \models \psi_1 \;\wedge\; \forall 0 \leq k \leq j.\, M, \pi[k, \infty] \models \psi_2)$
$M, \pi \models \psi_1 \wedge \psi_2$ *iff* $M, \pi \models \psi_1 \wedge M, \pi \models \psi_2$
$M, \pi \models \psi_1 \vee \psi_2$ *iff* $M, \pi \models \psi_1 \vee M, \pi \models \psi_2$

The model M *satisfies* **ACTL** *(state) formula* φ *(written* $M \models \varphi$*) iff* $M, s_0 \models \varphi$*, for all initial states* $s_0 \in I$*.*

2.2 Logic of Time and Strategies

In order to capture the notion of agents' strategic abilities (and thus be able to express many more subtle properties) we will use alternating-time temporal logics **ATL** and **ATL*** [AHK97; AHK02; Sch04].

There are many semantic variants of **ATL** and **ATL***; one of their key differences originates in the assumptions of what an agent can do (e.g., their memory, observation and other capabilities). For practical reasons[4], within this thesis we will focus on the variant of *incomplete information* (i.e., agents see/access only the part of the system state) and *imperfect recall* (i.e., agent's memory is modelled explicitly), which was originally proposed by Schobbens in [Sch04].[5]

Definition 2.12 (Syntax of **ATL**). *Given a finite set of agents* $\mathbb{A}\mathrm{gt}$ *and a set of atomic propositions* AP*, the syntax of* **ATL** *is defined by the following grammar:*

$$\phi ::= p \mid \neg\phi \mid \phi \vee \phi \mid \langle\langle A\rangle\rangle \mathrm{X}\psi \mid \langle\langle A\rangle\rangle \mathrm{G}\psi \mid \langle\langle A\rangle\rangle \psi \,\mathrm{U}\, \psi$$

where $p \in AP$ *is an atomic proposition,* $A \subseteq \mathbb{A}\mathrm{gt}$ *is a subset of agents (called coalition), temporal operators "*X*", "Always" and "*U*" stand for "in the next state", "always from now on" and "(strong) until" respectively.*

Additional Boolean connectives and temporal operators can be derived in a standard manner as previously demonstrated.

Informally, a formula $\langle\langle A\rangle\rangle\gamma$ says that a group of agents A can enforce γ no matter how other agents $\mathbb{A}\mathrm{gt}\backslash A$ proceed.

Below we introduce concepts of game structures and strategies that are necessary for the semantics of alternating-time temporal logic.

Definition 2.13 (CGS). *An imperfect information* ***concurrent game structure*** *(CGS) is a tuple* $M = (\mathbb{A}\mathrm{gt}, St, AP, L, Act, d, o, \{\sim_a \mid a \in \mathbb{A}\mathrm{gt}\})$*, where:*

- $\mathbb{A}\mathrm{gt} = \{1, \ldots, k\}$ *is a non-empty finite set of agents,*
- St *is a finite non-empty set of states,*
- AP *is a set of atomic propositions,*
- $L : St \mapsto 2^{AP}$ *is a labelling function,*
- Act *is a non-empty set of actions,*
- $d : \mathbb{A}\mathrm{gt} \times St \mapsto 2^{Act}$ *denotes actions that are available for each agent in each state,*
- $o : St \times Act^1 \times \ldots \times Act^k \mapsto St$ *is a transition function that assigns the outcome state* $q' = o(q, \alpha_1, \ldots, \alpha_k)$ *to each state* q *and tuple of actions* $\langle \alpha_1, \ldots, \alpha_k \rangle$*, such that* $\alpha_i \in d_i(q)$ *for* $i = 1, \ldots, k$*,*
- $\sim_a \subseteq St \times St$ *is an (epistemic) equivalence relation for each* $a \in \mathbb{A}\mathrm{gt}$*.*

Informally, whenever $q \sim_a q'$*, the states* q *and* q' *are said to be indistinguishable by an agent* a*. Here, every CGS* M *is assumed to be* ***uniform****, that is*

$$\forall_{q,q' \in St} \left(q \sim_a q' \quad \Rightarrow \quad d_a(q) = d_a(q') \right)$$

Remark 2.1. *A model from Definition 2.1 can be seen as a special case of CGS with a single agent only, over which satisfiability of both* **ATL** *and* **CTL** *formulae coincides [AHK02]. However, in the general case, CGS gives a more descriptive representation of the system behaviour, but is also less compact.*

Definition 2.14 (Strategy). *A memoryless* ***strategy*** *of an agent* $a \in \mathbb{A}\mathrm{gt}$ *is a function* $\sigma_a : St \mapsto Act^a$*, which species what an agent should do in a given state. The set of all strategies for* a *is denoted by* Σ_a^{ir}*.*

[4] In general, model checking of **ATL** (and thus **ATL***) with perfect recall and incomplete information is known to be undecidable.

[5] This variant is often labelled by $\mathbf{ATL}_{\mathrm{ir}}$ and $\mathbf{ATL}^*_{\mathrm{ir}}$, where the lower-case "i" stands for imperfect information, and the lower-case "r" stands for imperfect recall.

*A **collective strategy** of a group of agents $A = 1, \ldots, k \subseteq \mathbb{A}\mathrm{gt}$ is a tuple $\sigma_A = (\sigma_1, \ldots, sigma_k)$ of their individual strategies. The set of all strategies for group A is denoted by Σ_A^{ir}.*

Definition 2.15 (Outcome). *Given a state $q \in St$ and a group of agents $A \subseteq \mathbb{A}\mathrm{gt}$, the outcome function $out^{ir}(q, \sigma_A)$ returns a set of all paths that can occur in CGS M, when starting from the state q agents in A execute the strategy σ_A.*[6]

Definition 2.16 (Semantics of **ATL**). *Given a CGS M, state $q \in St$ and path $\lambda \in Paths(M)$, the satisfaction relation $\models$ is inductively defined as follows:*

$$
\begin{array}{ll}
M, q \models p & \text{iff } q \in L(p) \\
M, q \models \neg\phi & \text{iff } M, q \not\models \phi \\
M, q \models \phi_1 \vee \phi_2 & \text{iff } M, q \models \phi_1 \text{ or } M, q \models \phi_2 \\
M, q \models \langle\langle A \rangle\rangle \mathrm{X}\phi & \text{iff } \exists \sigma_A \in \Sigma_A^{\mathrm{ir}}, \text{ such that for all } \lambda \in out^{\mathrm{ir}}(q, \sigma_A).\ M, \lambda[1, \infty] \models \phi \\
M, q \models \langle\langle A \rangle\rangle \mathrm{G}\phi & \text{iff } \exists \sigma_A \in \Sigma_A^{\mathrm{ir}}, \text{ such that for all } \lambda \in out^{\mathrm{ir}}(q, \sigma_A).\ \forall i \geq 0.\ M, \lambda[i] \models \phi \\
M, q \models \langle\langle A \rangle\rangle \phi_1 \,\mathrm{U}\, \phi_2 & \text{iff } \exists \sigma_A \in \Sigma_A^{\mathrm{ir}}, \text{ such that for all } \lambda \in out^{\mathrm{ir}}(q, \sigma_A).\ \exists i \geq 0.\ M, \lambda[i] \models \phi_2 \\
 & \quad \text{and } \forall 0 \leq j < i.\ M, \lambda[j] \models \phi_1
\end{array}
$$

where $p \in AP$ and $A \subseteq \mathbb{A}\mathrm{gt}$.

The **ATL*** is an even more expressive logic that embeds "vanilla" **ATL**.

Definition 2.17 (Syntax of **ATL***). *Given a finite set of agents $\mathbb{A}\mathrm{gt}$ and a set of atomic propositions AP, the syntax of **ATL*** **state formulae** ϕ and **path formulae** ψ is defined by the following grammar:*

$$
\begin{aligned}
\phi &::= p \mid \neg\phi \mid \phi \vee \phi \mid \langle\langle A \rangle\rangle \psi \\
\psi &::= \phi \mid \neg\psi \mid \psi \vee \psi \mid \mathrm{X}\psi \mid \psi \,\mathrm{U}\, \psi
\end{aligned}
$$

where $p \in AP$ is an atomic proposition, $A \subseteq \mathbb{A}\mathrm{gt}$ is a subset of agents (called coalition), temporal operators "X" and "U" stand for "in the next state" and "(strong) until" respectively.

Additional Boolean connectives and temporal operators can be derived in a standard way as shown previously.

Definition 2.18 (Semantics of **ATL***). *Given a CGS M, state $q \in St$ and path $\lambda \in Paths(M)$, the satisfaction relation $\models$ is inductively defined as follows:*

$$
\begin{array}{ll}
M, q \models p & \text{iff } q \in L(p) \\
M, q \models \neg\phi & \text{iff } M, q \not\models \phi \\
M, q \models \phi_1 \vee \phi_2 & \text{iff } M, q \models \phi_1 \text{ or } M, q \models \phi_2 \\
M, q \models \langle\langle A \rangle\rangle \psi & \text{iff } \exists \sigma_A \in \Sigma_A^{\mathrm{ir}}.\ M, \lambda \models \psi, \text{for all } \lambda \in out^{\mathrm{ir}}(q, \sigma_A) \\
 & \\
M, \lambda \models \phi & \text{iff } M, \lambda[0] \models \phi \\
M, \lambda \models \neg\psi & \text{iff } M, \lambda \not\models \psi \\
M, \lambda \models \psi_1 \vee \psi_2 & \text{iff } M, \lambda \models \psi_1 \text{ or } M, \lambda \models \psi_2 \\
M, \lambda \models \mathrm{X}\psi & \text{iff } M, \lambda[1, \infty] \models \psi \\
M, \lambda \models \psi_1 \,\mathrm{U}\, \psi_2 & \text{iff } \exists i \geq 0.\ M, \lambda[i, \infty] \models \psi_2 \text{ and } M, \lambda[j, \infty] \models \psi_1, \text{for all } 0 \leq j < i
\end{array}
$$

where $p \in AP$ and $A \subseteq \mathbb{A}\mathrm{gt}$.

[6] The previously introduced notion of run and path applies to the case of CGS with no changes.

2.3 Adding Epistemic Operators

In this section we describe how the logics from Sections 2.1 and 2.2 can be extended with an epistemic dimension, allowing to reason about both time and knowledge [Jam15].[7]

The temporal-epistemic logic **CTLK** (reps. **CTLK***) can be seen as a syntactic variant of **CTL** (resp. **CTL***) with the addition of epistemic operator K.

Definition 2.19 (Syntax of **CTLK**)**.** *Given the set of agents* $\mathbb{A}\mathrm{gt}$ *and a set of atomic properties* AP*, the syntax of* **CTLK** *state formulae* φ *and path formulae* ψ *is given by the following grammar:*

$$\begin{aligned} \varphi &::= \mathsf{true} \mid p \mid \neg\varphi \mid \varphi \vee \varphi \mid \mathsf{A}\psi \mid K_a\varphi \\ \psi &::= \mathsf{X}\varphi \mid \varphi \,\mathsf{U}\, \varphi \end{aligned}$$

where $p \in AP$*,* $a \in \mathbb{A}\mathrm{gt}$ *and* K_a *is modal operator for knowledge. A formula* $K_a\varphi$ *reads as "agent* a *knows that* φ*". The clauses for Boolean connectives are standard; additional temporal operators "sometime" and "always" can be derived in the same way as for* **CTL***.*

Definition 2.20 (Syntax of **CTLK***)**.** *Given the set of agents* $\mathbb{A}\mathrm{gt}$ *and a set of atomic properties* AP*, the syntax of* **CTLK*** *state formulae* φ *and path formulae* ψ *is given by the following grammar:*

$$\begin{aligned} \varphi &::= \mathsf{true} \mid p \mid \neg\varphi \mid \varphi \vee \varphi \mid \mathsf{A}\psi \mid K_a\varphi \\ \psi &::= \varphi \mid \neg\psi \mid \psi \vee \psi \mid \mathsf{X}\psi \mid \psi \,\mathsf{U}\, \psi \end{aligned}$$

where $p \in AP$*,* $a \in \mathbb{A}\mathrm{gt}$ *and* K_a *is modal operator for knowledge. The clauses for Boolean connectives are standard; additional temporal operators "sometime" and "always" can be derived in the same way as for* **CTL****.*

Definition 2.21 (Extended Model)**.** *The extended model is a model* M *supplied with additional indistinguishability relation* $\sim_i \subseteq St \times St$*, one per agent* $i \in \mathbb{A}\mathrm{gt}$*.*

Remark 2.2. *Note that a model from Definition 2.1 can be seen as a special case of extended model with one agent only and indistinguishability relations given by identity relation over states.*

Definition 2.22 (Semantics of **CTLK** and **CTLK***)**.** *The semantics of* **CTLK** *and* **CTLK*** *are based on extended model; the semantic clauses correspond to the union of the following rule and those from* **CTL** *(see Definition 2.5) and* **CTL*** *(see Definition 2.7) respectively:*

$$M, s \models K_a\varphi \quad \textit{iff} \quad M, s' \models \varphi, \textit{for all } s' \sim_a s$$

Analogously, the standard **ATL** and **ATL*** can be further extended to support reasoning about agents' knowledge.

Definition 2.23 (Syntax of **ATLK**)**.** *Given a finite set of agents* $\mathbb{A}\mathrm{gt}$ *and a set of atomic propositions* AP*, the syntax of* **ATLK** *is defined by the following grammar:*

$$\phi ::= p \mid \neg\phi \mid \phi \vee \phi \mid \langle\langle A\rangle\rangle \mathsf{X}\psi \mid \langle\langle A\rangle\rangle \mathsf{G}\psi \mid \langle\langle A\rangle\rangle \psi \,\mathsf{U}\, \psi \mid K_a\psi$$

[7] Within this thesis we will focus on K, but in general, more kinds of epistemic operators (such as common knowledge, distributed knowledge, and mutual knowledge) may be introduced.

where $p \in AP$ is an atomic proposition, $A \subseteq \mathbb{A}\mathrm{gt}$ is a subset of agents (called coalition), temporal operators "X", "Always" and "U" stand for "in the next state", "always from now on" and "(strong) until" respectively, and "K_a" is modal operator for knowledge.

Definition 2.24 (Syntax of **ATLK***)**.** *Given a finite set of agents $\mathbb{A}\mathrm{gt}$ and a set of atomic propositions AP, the syntax of* **ATL*** ***state formulae*** ϕ *and* ***path formulae*** ψ *is defined by the following grammar:*

$$\phi ::= p \mid \neg\phi \mid \phi \vee \phi \mid \langle\langle A \rangle\rangle \psi$$
$$\psi ::= \phi \mid \neg\psi \mid \psi \vee \psi \mid \mathrm{X}\psi \mid \psi \,\mathrm{U}\, \psi$$

where $p \in AP$ is an atomic proposition, $A \subseteq \mathbb{A}\mathrm{gt}$ is a subset of agents (called coalition), temporal operators "X" and "U" stand for "in the next state" and "(strong) until" respectively, and "K_a" is modal operator for knowledge.

Definition 2.25 (Semantics of **ATLK** and **ATLK***)**.** *The semantics of* **ATLK** *and* **ATLK*** *require adding the following rule to the existing clauses from* **ATL** *(see Definition 2.16) and* **ATL*** *(see Definition 2.18) respectively:*

$$M, q \vDash K_a\phi \quad \textit{iff} \quad \forall_{q' \in St}(q \sim_a q' \Rightarrow M, q' \vDash \phi)$$

Remark 2.3. *We will sometimes explicitly write $\vDash_x$, where $x \in \{$***CTL***, ***CTL****, ***ACTL***, ***ACTL****, ***CTLK***, ***CTLK****, ***ATL***, ***ATL****, ***ATLK***, ***ATLK**** $\}$, when the implied logic is not clear from the context. Analogously, when the model (resp. CGS) M is clear from the context, we will often use a shorter notation $s \models \varphi$ and $\pi \models \psi$ (resp. $q \models \phi$ and $\lambda \models \psi$) instead of $M, s \models \varphi$ and $M, \pi \models \psi$ (resp. $M, q \models \phi$ and $M, \lambda \models \psi$).*

2.4 UPPAAL Model Checker

We will now briefly describe the UPPAAL model checker, which was the main tool used for modelling and verification of the forthcoming experimental cases. For more details, please refer to the official documentation [Upp02; BDL04].

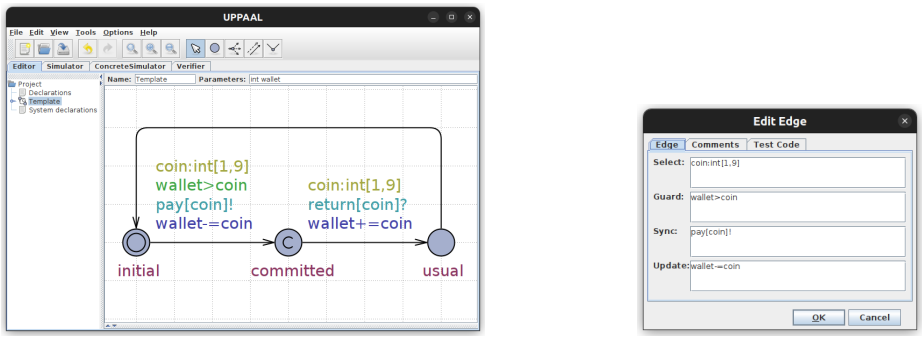
In UPPAAL, a system is represented by a parameterized network of finite automata (or processes).[8] Processes are instantiated from the templates by assigning template parameter(s) with the value(s). Templates contain a finite number of locations and labelled edges, which determine the local transition relations. Fig. 2.1 shows an example of different location types and labelled edges.

Locations are depicted by circles and represent the local states of the module. *Initial* locations are marked by a double circle. *Committed* locations are marked by circled 'C'. If any process is in a committed location, then the next transition must involve an edge from one of the committed locations. Those are used to create atomic sequences or encode synchronization between more than two components.

Edges define the local transitions in the module. They are annotated by selections (in yellow), guards (green), synchronizations (teal), and updates (blue). The syntax of expressions mostly coincides with that of C/C++. Note, that in UPPAAL syntax operators "=" and "==" represent assignment and equality check respectively.

[8] The original definition refers to extended timed automata. However, associated functionalities present little to no interest in the context of this work and are not used. Entailed differences are insignificant, and further definitions are adapted in accordance with our usage scenario.

A *select* label nondeterministically binds an identifier to a value from the range. A *guard* is side-effect free expression evaluating to boolean, a pre-condition for edge to be possibly taken; when the guard condition evaluates to true, the edge is said to be (locally) *enabled*. *Synchronisation* allows processes to synchronize over a common channel (appended with `!` or `?` for sender and receiver resp.). In the presence of a synchronisation label, an edge can only be "fired" together with a complementary one in another process. An *update* is a comma-separated sequence of expressions (mainly assignment statements), which are evaluated when the edge is taken. For synchronous edges, the sender's update expression takes precedence over that of the receiver. Straightforward value passing over a channel is not allowed; instead, one has to use shared global variables for the transmission.



(A) Locations and edge labels (B) Edge label components

FIGURE 2.1: UPPAAL locations and edges mock-up example.

Right up to the end of the thesis, we will treat agent graphs, agent types and MAS graphs as processes, templates and automata networks respectively.

The system requirements in UPPAAL are specified using a branching-time logic **CTL*** [Eme90]. The syntax of supported **CTL*** (state) formulas over a set of atomic propositions AP is given by the following grammar:

$$\varphi \quad ::= \quad \mathsf{A}\mathsf{F}p \mid \mathsf{A}\mathsf{G}p \mid \mathsf{E}\mathsf{F}p \mid \mathsf{E}\mathsf{G}p \mid \mathsf{A}\mathsf{G}(p \Rightarrow \mathsf{A}\mathsf{F}q)$$

where A and E are universal and existential path quantifiers, G and F are temporal operators for "always" and "sometime", and $p, q \in AP$ are atomic properties. Note that a set of atomic properties AP consists of all agent locations and logical expressions over MAS variables and literals from their domains.

In order to check if **CTL*** formula φ is satisfied by a given system, UPPAAL verifier incrementally *unfolds* underlying MAS graph into a labelled transition system (LTS), where each *state* corresponds to a tuple of current locations, one per agent, and evaluation of the variables. However, contrary to standard practice to interpret satisfaction relation $\models$ only over maximal paths — sequences of states, connected with transitions, that are either infinite or end in a deadlock state with no outgoing transitions — UPPAAL admits *all non-empty finite paths*. This leads to formulas AF and EG being less useful in practice, usually requiring extra gimmicks to filter out unwanted "counter-example" paths.

Example 2.1. *Consider a toy example portraying the variant of the Toads&Frogs Puzzle [BCG04], where N(=3) toads and M(=3) frogs are placed on N-leftmost and M-rightmost tiles of $N + M + 1$(=7) by 1 size board. Each tile can only be occupied by one animal, toads can only move rightwards, and frogs leftwards. There are two possible moves:* ***slide*** *to the*

*next (unoccupied) tile or **jump** over another animal to the two-tiles apart (unoccupied) spot. The question is whether the animals can switch their positions, that is toads move to the N-rightmost positions and frogs move to the M-leftmost positions.*

The Fig. 2.2 shows the Editor tab with graph representation of agent templates for the frog (right) and toad (left). Both templates are parameterized with an identifier, which determines an initial position for each respective instance. The global variable `occupied` *is the $(N + M + 1)$-size list of Booleans that says if the respective tile is empty. Each instance of animal has a local variable* `pos` *that is a bounded integer (with a range of values from 0 to $N + M$) representing its position.*

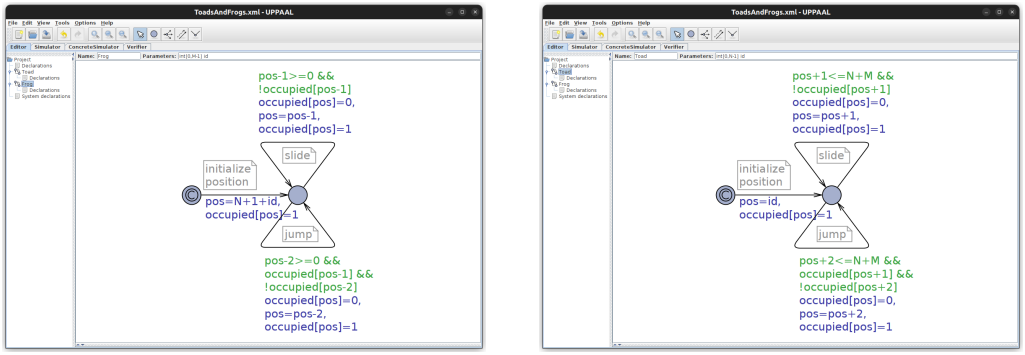
The Fig. 2.3 depicts the simulator tab, where a possible execution path can be manually composed and inspected, and the verifier tab, where the queries representing the system properties are checked. The winning condition is captured by the following **CTL*** *formula:*

$$\mathsf{EF}(\bigwedge_{i=1..N} Toad_i > mid) \wedge (\bigwedge_{j=1..M} Frog_j < mid)$$

where propositions $Toad_i$ and $Frog_j$ denote the positions of respective animals, and $mid = M + 1$ is an index of the middle (empty) cell, which initially separated two types of animals. By design of the model already embeds the requirement that no pair of animals might occupy the same cell simultaneously. This formula naturally translates into the following UPPAAL *query:*

```
E<>(forall(i:int[0,N-1])Toad(i).pos>M) &&
   (forall(j:int[0,M-1])Frog(j).pos<M)
```

Note that in the model, due to starting the indexing from zero, the range of iterators and the middle cell index were all shifted down by one.

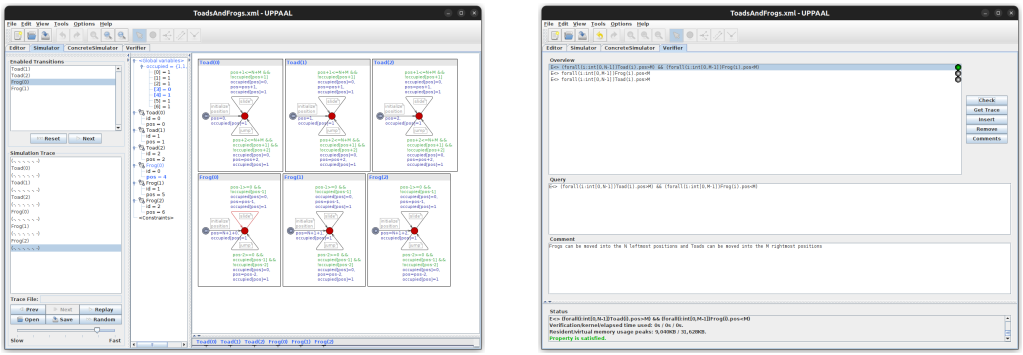


(A) Frog template (B) Toad template

FIGURE 2.2: Toads&Frogs Puzzle in UPPAAL (Part 1/2).

2.5 Related Work

Voting schemes and their properties were analysed using many various — often conceptually different — approaches. Intuitively, their key differences could be based on: the representation of a system, specification of properties, considered threats and chosen verification/reasoning form/paradigm (incl. assumptions it relies on, kind of guarantees that are promised and other attributes). Each method has its strengths and limitations, and, with exception to the pen-and-paper based, may



(A) Simulator view (B) Verifier view

FIGURE 2.3: Toads&Frogs Puzzle in UPPAAL (Part 2/2).

have an implementation in the form of a software package or a toolbox. As was pointed out in [Che+22], selecting the "right" tool for a given problem can be quite challenging even for experienced users, as it often requires expert knowledge with a good understanding of internal technicalities. This section will attempt to present a high-level overview selected number of such methods on examples of several studies of voting protocols, including a few recent ones.

In [KR05] authors utilize an *applied pi calculus* ([AF01]) formalism to model election protocol FOO 92 ([FOO93]), they formalise its three expected properties: fairness, eligibility and privacy, and jointly use an automated tool PROVERIF as well as manual proofs to verify those. The system behaviour is represented as a combination of concurrent processes (e.g., environment, agent roles: voter, administrator, etc.) and their interactions. The properties are modelled as un-/reachability properties, which say that some (goal) can/cannot be reached, and/or in terms of observational equivalences, saying e.g. that intruder/potential should be unable to distinguish between two certain processes. The verification of the latter is incomplete in PROVERIF, and thus some manual proofs are provided. Interestingly, the authors note that during the modelling an implicit (or insufficiently stressed) assumption on participants' behaviour was revealed that happens to be critical for privacy to hold.

The series of work by Zollinger et al. [Zol20] utilize TAMARIN to for verification of the tracker-based schemes like Selene. Privacy-related proofs are conducted using observation equivalence. For example, ballot privacy (BP) could be expressed as the indistinguishability of two (or more) system executions, where voters' votes were swapped (or permuted). Authors also study usability, which refers to the quality of users' experience and ability to carry out their tasks/goals efficiently. They introduce the mental model for evaluating the level of voters' understanding and trust, apply that for case study of their Selene implementation and discuss some interesting findings. In particular, they investigate the link between trust and understanding and propose a new voting-oriented metric of trust.

In [Che+22] authors propose a new formalism SAPIC$^+$ intended as "swiss-army knife" combining/joining TAMARIN, PROVERIF and DEEPSEC together under the unified input specification. The SAPIC$^+$ proposes a specification language, which is based on applied pi-calculus, and proven to be correct translations for each tool's original specification syntax. Not only does it combine the strengths of these tools individually, but also facilitates a better understanding of the system and higher assurance in the correctness of results.

Unfortunately, there exists no reliable analogue solution for other, broader scope of tools (despite some notable attempts, e.g. by developers of LTSMIN toolbox, where translations support quite limited fragment of original syntax and provided documentation is often poor or obsolete, or cite-other-experimental-translators).

More extensive discussion of related work can be found in respective sections.

Chapter 3

Towards Model Checking of Voting Protocols in Uppaal

The design and implementation of a trustworthy e-voting system is a challenging task. Formal analysis can be of great help here. In particular, it can lead to a better understanding of how the voting system works, and what requirements on the system are relevant. In this chapter, we propose that the state-of-the-art model checker UPPAAL provides a good environment for modelling and preliminary verification of voting protocols. To illustrate this, we demonstrate how to model a version of Prêt à Voter in UPPAAL, together with some natural extensions. We also show how to verify a variant of receipt-freeness, despite the severe limitations of the property specification language in the model checker.

The aim of this work is to open a new path, rather than deliver the ultimate outcome of formal analysis.

3.1 Introduction

The design and implementation of a good e-voting system is highly challenging. Real-life systems are notoriously complex and difficult to analyze. Moreover, elections are *social* processes: they are run by humans, with humans, and for humans, which makes them unpredictable and hard to model. Last but not least, it is not always clear what *good* means for a voting system. A multitude of properties have been proposed by the community of social choice theory (such as Pareto optimality and non-manipulability), as well as researchers who focus on the security of voting (cf. ballot secrecy, coercion-resistance, voter-verifiability, and so on). The former kind of properties are typically set for a very abstract view of the voting procedure and consequently miss many real-life concerns. For the latter, it is often difficult to translate the informal intuition to a formal definition that will be commonly accepted.

In a word, we deal with processes that are hard to understand and predict, and seek to evaluate them against criteria for which we have no clear consensus. Formal analysis can be of great help here: perhaps not in the sense of providing the ultimate answers, but rather to strengthen our understanding of both how the voting system works and how it should work. The main goal of this work is to propose that model checkers from distributed and multi-agent systems can be invaluable tools for such an analysis.

Model checkers and UPPAAL. Much research on model checking focuses on the design of logical systems for a particular class of properties, establishing their theoretical characteristics, and development of verification algorithms. This obscures the fact that a model checking framework is valuable as long as it is actually *used* to analyze something. The analysis does not have to result in a "correctness certificate" of the system under scrutiny. A readable model of the system and an understandable formula capturing the requirement are already of substantial value.

In this context, two features of a model checker are essential. On the one hand, it should provide a *flexible model specification language* that allows for modular and succinct specification of processes. On the other hand, it must offer a *good graphical user interface*. Paradoxically, tools satisfying both criteria are rather scarce. Here, we suggest that the state-of-the-art model checker UPPAAL can provide a nice environment for modelling and preliminary verification of voting protocols and their social context. To this end, we show how to use UPPAAL to model a voting protocol of choice (in our case, a version of Prêt à Voter), and to verify some requirements written in the temporal logic **CTL**.

Contribution. The main contribution of this work is methodological: we demonstrate that specification frameworks and tools from distributed and multi-agent systems can be useful in the analysis and validation of voting procedures. An additional, technical contribution consists in a reduction from model checking of temporal-epistemic specifications to purely temporal ones, in order to verify a variant of receipt-freeness despite the limitations of UPPAAL.

We emphasize that this is a preliminary work, aimed at exploring a path rather than delivering the ultimate outcome of formal analysis. A comprehensive model of Prêt à Voter, more accurate specification of requirements, and exhaustive verification are planned for the future. We also plan to cover social engineering-style attacks involving interactions between coercers (or vote-buyers) and voters. This will require, however, a substantial extension of the algorithms in UPPAAL or a similar model checker.

Structure of the chapter. We begin by introducing the main ideas behind modelling and model checking of multi-agent systems, including a brief introduction to UPPAAL (Section 3.2). In Section 3.3, we provide an overview of Prêt à Voter, the voting protocol that we will use for our study. Section 3.4 presents a multi-agent model of the protocol; some interesting extensions of the model are proposed in Section 3.6. We show how to specify simple requirements on the voting system, and discuss the output of model checking in Section 3.5. The section also presents our main technical contribution, namely the model checking reduction that recasts knowledge-related statements as temporal properties. We discuss related work in Section 3.7, and conclude in Section 3.8.

3.2 Towards Model Checking of Voting Protocols

Model checking is the decision problem that takes a model of the system and a formula specifying correctness, and determines whether the model satisfies the formula. This allows for a natural separation of concerns: the model specifies how the system is, while the formula specifies how it should be. Moreover, most model checking approaches encourage systematic specification of requirements, especially for the requirements written in modal and temporal logic. In that case, the behaviour of the system is represented by a transition network, possibly with additional modal relations to capture e.g. the uncertainty of agents. The structure of the network is typically given by a higher-level representation, e.g., a set of agent templates together with a synchronization mechanism.

We begin with a brief recall of UPPAAL, the model checker that we will use in later sections. A more detailed introduction can be found in Section 2.4 or its official documentation [BDL04].

3.2.1 Modelling in UPPAAL

An UPPAAL model consists of a set of concurrent processes. The processes are defined by templates, each possibly having a set of parameters. The parameterized templates are used for defining a large number of almost identical processes. Every template consists of *nodes*, *edges*, and optional local declarations. An example template is shown in Figure 3.2; we will use it to model the behaviour of a voter.

For convenience, we will place the selections and guards at the top or left of an edge, and the synchronizations and updates at the bottom/right.

3.2.2 Specification of Requirements

To specify requirements, UPPAAL uses a fragment of the temporal logic **CTL** [Eme90]. **CTL** allows for reasoning about the possible execution paths of the system by means of the *path quantifiers* E ("there is a path") and A ("for every path"). A path is a maximal[1] sequence of states and transitions. To address the temporal pattern on a path, one can use the *temporal operators* X ("in the next moment"), G ("always from now on"), F ("now or sometime in the future"), and U ("until"). For example, the formula $\mathsf{AG}\big(\mathsf{has\_ballot_i} \Rightarrow \mathsf{AF}(\mathsf{voted_{i,1}} \vee \cdots \vee \mathsf{voted_{i,k}})\big)$ expresses that, on all paths, whenever voter i gets her ballot form, she will eventually cast her vote for one of the candidates $1, \ldots, k$. Another formula, $\mathsf{AG}\neg\mathsf{punished_i}$ says that voter i will never be punished by the coercer.

[1] I.e., infinite or ending in a state with no outgoing transitions.

(a)

Discard	Retain
Obelix	
Idefix	
Asterix	
Panoramix	
	7304944

(b)

Retain
X
7304944

FIGURE 3.1: (a) Prêt à Voter ballot form; (b) Receipt encoding a vote for "Idefix".

More advanced properties usually require a combination of temporal modalities with *knowledge operators* K_a, where $K_a\phi$ expresses "agent a knows that ϕ holds." For example, formula $\mathsf{EF}(\mathsf{results} \wedge \neg \mathsf{voted}_{i,j} \wedge \neg K_c \neg \mathsf{voted}_{i,j})$ says that the coercer c might not know that voter i hasn't voted for candidate j, even if the results are already published. Moreover, $\mathsf{AG}(\mathsf{results} \Rightarrow \neg K_c \neg \mathsf{voted}_{i,j})$ expresses that, when the results are out, the coercer won't know that the voter refused to vote for j. Intuitively, both formulas capture different strength of receipt-freeness for a voter who has been instructed to vote for candidate j.

3.3 Outline of Prêt à Voter

In this work, we use UPPAAL for modelling and analysis of a voting protocol. The protocol of choice is a version of Prêt à Voter. We stress that this is not an up-to-date version of Prêt à Voter but it serves to illustrate how some attacks can be captured with UPPAAL. A short overview of Prêt à Voter is presented here; the full details can be found, for example, in [Rya10] or [HR16].

Most voter-verifiable voting systems work as follows: at the time of casting, an encryption or encoding of the vote is created and posted to a secure public bulletin board (BB). The voter can later check that her encrypted ballot appears correctly. The set of posted ballots is then processed in some verifiable way to reveal the tally or outcome. Much of this is effectively a secure distributed computation, and as such is well-established and understood in cryptography. The really challenging bit is the creation of the encrypted ballots because it involves interactions between the users and the system. This has to be done in a way that assures the voter that her vote is correctly embedded, while avoiding introducing any coercion or vote-buying threats.

The key innovation of the Prêt à Voter approach is to encode the vote using a randomised candidate list. This contrasts with earlier verifiable schemes that involved the voter inputting her selection to a device that then produces an encryption of the selection. Here what is encrypted is the candidate order which can be generated and committed in advance, and the voter simply marks her choice on the paper ballot in the traditional manner.

Suppose that our voter is called Anne. At the polling station, Anne is authenticated and registered and she chooses at random a ballot form sealed in an envelope and saunters over to the booth. An example of such a form is shown in Figure 3.1a. In the booth, she extracts her ballot form from the envelope and marks her selection in the usual way by placing a cross in the right-hand column against the candidate of her choice (for approval or ranked voting, she marks her selection or ranking against the candidates). Once her selection has been made, she separates the left and right hand strips along a thoughtfully provided perforation and discards the left-hand

strip. She keeps the right-hand strip which now constitutes her *privacy protected receipt*, as shown in Figure 3.1b.

Anne now exits the booth clutching her receipt, returns to the registration desk, and casts the receipt: it is placed over an optical reader or similar device that records the string at the bottom of the strip and registers which cells are marked. Her original paper receipt is digitally signed and franked and returned to her to keep and later check that her vote is correctly recorded on the bulletin board. The randomisation of the candidate list on each ballot form ensures that the receipt does not reveal the way she voted, thus ensuring the secrecy of her vote. Incidentally, it also removes any bias towards the candidate at the top of the list that can occur with a fixed ordering.

The value printed on the bottom of the receipt is what enables extraction of the vote during the tabulation phase: buried cryptographically in this value is the information needed to reconstruct the candidate order and so extract the vote encoded on the receipt. This information is encrypted with secret keys shared across a number of tellers. Thus, only a threshold set of tellers acting together are able to interpret the vote encoded in the receipt. In practice, the value on the receipt will be a pointer (e.g. a hash) to a ciphertext committed to the bulletin board during the setup phase.

After the voting phase, voters (or perhaps proxies acting on their behalf) can visit the Bulletin Board and confirm that their receipts appear correctly. Once any discrepancies are resolved, the tellers take over and perform anonymising mixes and decryption of the receipts. At the end, the plaintext votes will be posted in secret shuffled order, or in the case of homomorphic tabulation, the final result is posted. All the processing of the votes can be made universally verifiable, i.e., any observer can check that no votes were manipulated.

Prêt à Voter brings several advantages in terms of privacy and dispute resolution. Firstly, it avoids side channel leakage of the vote from the encryption device. Secondly, it improves on dispute resolution: ballot assurance is based on random audits of the ballot forms, which can be performed by the voter or independent observers. A ballot form is either well-formed, i.e. the plaintext order matches the encrypted order, or not. This is independent of the voter or her choice, hence there can be no dispute as to what choice the voter provided. Such disputes can arise in Benaloh challenges and similar cut-and-choose style audits. Furthermore, auditing ballots does not impinge on ballot privacy, as nothing about the voter or the vote can be revealed at this point.

3.4 Modelling Prêt à Voter in UPPAAL

In this section, we present how the components and participants of Prêt à Voter can be modelled in UPPAAL. To this end, we give a detailed description of each module template, its elements, and their interactions.[2] The templates represent the behaviour of the following types of agents: *voters*, *coercers*, *mix tellers*, *decryption tellers*, *auditors*, and the *voting infrastructure*. For more than one module of a given type, an identifier $i = 0, 1, \ldots$ will be associated with each instance.

The code of the model is available at http://tinyurl.com/pret-a-voter-model. Details of cryptographic primitives can be found in textbooks, e.g. [Sti05].

In the model, we use the *ElGamal* encryption algorithm, which allows for re-encryption. The public key is a tuple (p, α, β), where α is a generator of group $\mathbb{Z}_p^*$, $\beta = \alpha^k$ and k is a secret (private key). A plain-text m encrypted with public key PK with randomness y will be noted as $E_{PK}(m, y)$, if the value of y is not known, then

[2] In this chapter, we will use the terms "module" and "agent" interchangeably.

it will be noted as $E_{PK}(m, *)$ (or simply $E_{PK}(m)$) instead. A ciphertext c decrypted with private key K is noted by $D_K(c)$.

3.4.1 Environment

We begin with an overview of the shared environment of action, i.e., the data structures and variables shared by the modules. To capture a possibly repeated block of atomic update expressions, some procedures are introduced. This will allow for more complex expressions and a shorter, reader-friendly form of labels.

The environment includes some global read-only configuration variables:

- `c_total[=3]`: the number of candidates,
- `v_total[=3]`: the number of voters,
- `mt_total[=3]`: the number of mix tellers,
- `dt_total[=3]`: the number of decryption tellers,
- `dt_min[=2]`: the number of participants needed to reconstruct a secret key,
- `z_order[=7]`: an order of cyclic group $\mathbb{Z}_p^*$,
- `pk[=(3,6)]`: the pair of generator α of group $\mathbb{Z}_p^*$ and $\beta = \alpha^k \ (mod\ p)$, where k is a secret key.

From the model configuration values, we derive some auxiliary variables, such as lists of permutations of the batch terms `P_b`, list of permutations of the candidates `P_c`, list of cyclic shifts of the candidates `S_c`, lookup table `dlog`, that maps onion to its seed ($\pm$ possible candidate choice), list of combinations to select a subset of the batch intended for audit `audit_ch`, and list of ways of splitting that in two (odd and even) `audit_lr`. We precompute their values for a given configuration and declare them as constant type. To facilitate the readability and manageability of the model code, we define some data structures and type name aliases based on the configuration variables:

- **Ciphertext**: a pair (y_1, y_2). For the simplicity of modelling, we assume that ElGamal encryption is used.
- **Ballot**: a pair (θ, cl) of onion $\theta = E_{PK}(s, *)$ and candidate list $cl = \pi(s)$, where s is a seed associated with the ballot, and $\pi : \mathbb{R} \to Perm_C$ is a function that associates a seed with a permutation of the candidates. To allow absorption of the index of a marked cell into the onion, we use cyclic shifts of the base candidate order. This means that we just have simple ElGamal ciphertexts to mix.
- **Receipt**: a pair (θ, r) of onion θ and an index r of marked cell. It can be used to verify if a term was recorded and if it was done correctly.
- **c_t**: an integer with range [0, c_total), a candidate;
- **v_t**: an integer with range [0, v_total), a voter;
- **z_t**: an integer with range [0, z_total), an element of $\mathbb{Z}_p^*$.

The writable global variables include:

- `board`: a 2-dimensional list of ciphertexts, representing the web bulletin board. The first column is reserved for the batch of onions with absorbed indices from the receipt. The next $(mt\_total \cdot 2)$-columns store re-encryption mixes. The remaining $(dt\_min - 1)$-columns store the intermediate results of threshold decryption; the last one holds the decrypted message.
- `mixes`: encodes which mix teller has a turn at the moment;
- `dt_curr`: encodes the number of currently participating decryption tellers;
- `decryptions`: the number of decryptions made.

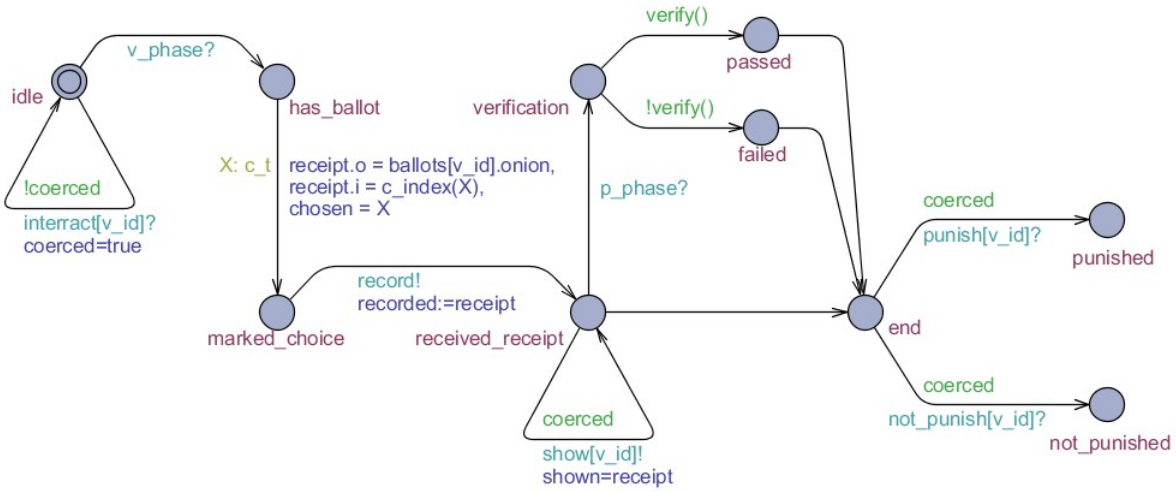


FIGURE 3.2: Voter template for the model of Prêt à Voter.

There is also a set of globally shared variables used to simulate the passing of a value through a channel; their use will be specified in transition descriptions.

3.4.2 Voter Template

The structure of the Voter template is shown in Figure 3.2. The idea is that while the voter waits for the start of an election she might be subject to coercion. When the ballots are ready, the voter selects a candidate and transmits the receipt to the system. Then she decides if she wants to check how her vote has been recorded and if she wants to show the receipt to the coercer. If coerced, she also waits for the coercer's decision to punish her or refrain from punishment. The module includes the following private variables:

- `receipt`: an instance of `Receipt`, obtained after casting a vote;
- `coerced[=false]`: a Boolean value, indicating if coercer has established a contact;
- `chosen`: integer value of the chosen candidate.

Moreover, the following procedures are included:

- `c_index(target)`: returns an index, at which `target` can be found on the candidate list of a ballot;
- `verify()`: returns *true* if the voter's `receipt` can be found on the Web Bulletin Board, else it returns $false$.

Local states:

- *idle*: waiting for the election, might get contacted by coercer;
- *has_ballot*: the voter has already obtained the ballot form;
- *marked_choice*: the voter has marked an index of the chosen candidate (and destroyed left-hand side with candidate list);
- *received_receipt*: the receipt is obtained and might be shown to the coercer;
- *verification*: the voter has decided to verify the receipt;
- *passed*: the voter got a confirmation that the receipt appears correctly;
- *failed*: the voter obtains evidence that the receipt does not appear on BB or appears incorrectly (in case of index absorption both cases are the same);
- *end*: the end of the voting ceremony;

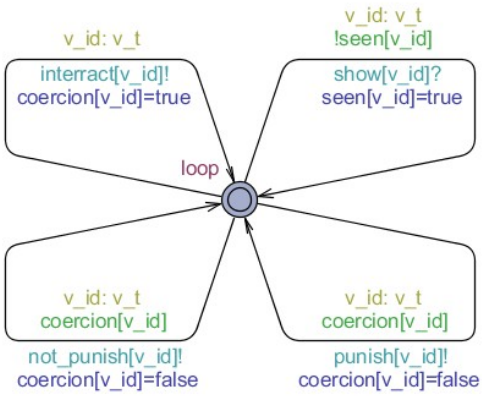


FIGURE 3.3: Coercer template.

- *punished*: the voter has been punished by the coercer;
- *not_punished*: the coercer refrained from punishing the voter.

Transitions:

- *idle*→*idle*: if was not already coerced, enable transition; if taken, then set `coercion` to *true*;
- *idle*→*has_ballot*: always enabled; if taken, the voter acquires a ballot form;
- *has_ballot*→*marked_choice*: mark the cell with the selected candidate;
- *marked_choice*→*received_receipt*: send `receipt` to the Sys process over channel `record` using shared variable `recorded`;
- *received_receipt*→*received_receipt*: if was coerced, enable transition; if taken, then pass the `receipt` to the coercer using shared variable `shown`;
- *received_receipt*→*verification*: always enabled; if taken, the voter decides to verify whether the receipt appears on the BB;
- *(received_receipt* || *passed* || *failed)*→*end*: a voting ceremony ends for the voter;
- *end*→*punished*: if was coerced, enable transition; if taken, then the voter has been punished by the coercer;
- *end*→*not_punished*: if was coerced, enable transition; if taken, the coercer has refrained from punishing the voter.

3.4.3 Coercer

The coercer can be thought of as a party that tries to influence the outcome of the vote by forcing voters to obey certain instructions. To enforce this, the coercer can punish the voter, or refrain from the punishment. The structure of the Coercer module is presented in Figure 3.3.

Private variables:

- `coercion`: a Boolean list used to keep track of the voters that have been coerced;
- `seen`: a Boolean list, that indicates if the voter has shown proof of her vote.

There is only one state called *loop* for Coercer. It has 4 looping transitions. Their update expressions take the form of a Boolean value assignment. A more common modelling approach could be to clone the local state for possible Boolean evaluations and find a reachable subset there. However, this would lead to a loss of generality and readability of the module (in particular, resulting in the coercer module having 2^{2n} local states for n voters).

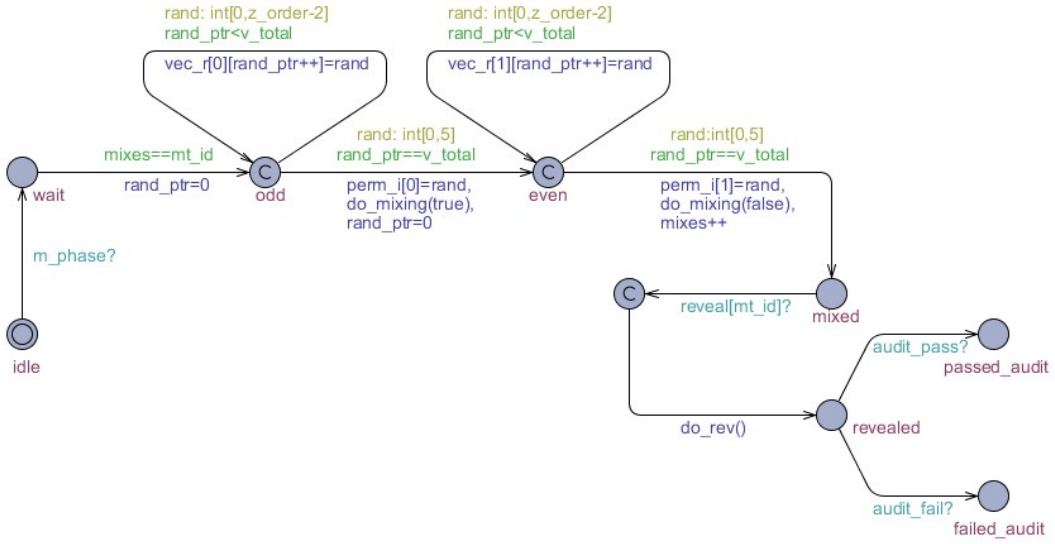


FIGURE 3.4: Mteller template.

Local transitions synchronizing with voter `v_id` (listed clockwise starting from top-left corner):

- *loop*→*loop* (top-left): establish contact with the voter, set `coercion[v_id]` to *true*;
- *loop*→*loop* (top-right): if have not seen proof of vote, enable transition; if taken, set `seen[v_id]` to *true*;
- *loop*→*loop* (bottom-right): if the voter was coerced, enable transition; if taken, then punish the voter, set `coercion[v_id]` to $false$, finalizing interaction;
- *loop*→*loop* (bottom-left): if the voter was coerced, enable transition; if taken, then do not punish the voter, set `coercion[v_id]` to $false$, finalizing interaction.

3.4.4 Mix Teller (Mteller)

Once the mixing phase starts, each mix teller performs two re-encryption mixes. The order of turns is ascending and determined by their identifiers. The randomization factors and permutation of each mix are selected in a nondeterministic way and stored for a possible audit of re-encryption mixes. When audited, the mix teller reveals the requested links and the associated factors, thus allowing the Auditor to verify that the input ciphertext maps to the output. The structure of the mix teller is shown in Figure 3.4.

Private variables:

- `vec_r`: 2-dimensional integer list (size $2 \times |v\_total|$) of randomization factors used for re-encryption;
- `perm_i`: 2-dimensional integer list (size $2 \times |v\_total|$) of permutation indices used for re-encryption;
- `rand_ptr`: index for `vec_r`;
- `mycol`: a pair of `board` column indices reserved for a given mix teller.

Procedures:

- `do_mixing(mi)`: using board column `(mycol[mi]-1)` as an input, re-encrypt each term using randomization factors from `vec_r[mi]`, shuffle them with `perm_i[mi]` permutation and paste result to `mycol[mi]`;

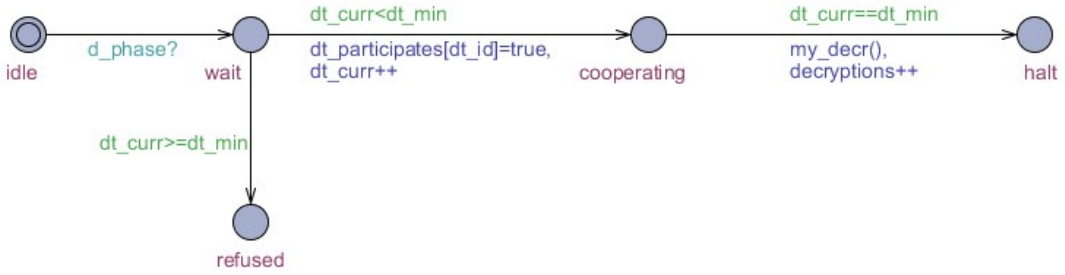


FIGURE 3.5: Dteller template.

- `do_rev()`: uses shared variables *rev_r* and *rev_p* to reveal (pass to the Auditor) randomization factors and links for the audited terms.

Local states:

- *idle*: waiting for the start of the mixing phase;
- *wait*: waiting for a turn for mixing;
- *odd*: performing odd mix;
- *even*: performing even mixing;
- *mixed*: finished mixing, passed the turn to the next mix teller (if any), waiting for a possible audit;
- *revealed*: revealed values needed for audit, waiting for an Auditor's verdict;
- *passed_audit*: mix teller passed Auditor's correctness check;
- *failed_audit*: mix teller failed Auditor's correctness check.

Local transitions:

- *idle*→*wait*: enabled transition;
- *wait*→*odd*: if it is the current mix teller's turn, enable transition; if taken, then initialize `rand_ptr` to zero;
- *odd*→*odd*: if new randomization factors were not selected yet, enable transition; if taken, insert random value *rand* into `vec_r[0][rand_ptr]` and then increment `rand_ptr`;
- *odd*→*even*: if randomization factors are ready, enable transition; randomly select permutation index, perform re-encryption mix using those and reset `rand_ptr` counter to generate a new randomness;
- *even*→*even*: if new randomization factors were not selected yet, enable transition; if taken, insert random value *rand* into `vec_r[1][rand_ptr]` and then `rand_ptr`;
- *even*→*mixed*: if randomization factors are ready, enable transition; randomly select permutation index, perform re-encryption mix using those and pass turn incrementing `mixes`;
- *mixed*→*audit*: enabled transition; if taken, then mix teller will be in a committed state, from where will have to reveal the mix factors for audited terms;
- *revealed*→*(passed_audit* `||` *failed_audit)*: enabled transition.

3.4.5 Decryption Teller (Dteller)

In this module, after the re-encryption mixes are done, a subset of cooperating decryption tellers is chosen nondeterministically. Note that if a subset has less than two elements (e.g. when two or more decryption tellers refused to cooperate), then

they should not be able to reconstruct a secret key, which would lead to a deadlock. In order to avoid that, only subsets with cardinality of 2 are considered in our simplified model.

We use *Shamir (t,w)-Threshold Scheme* for decryption, with $t = 2$ and $w = 3$. Consider a polynomial $a(x) = a_0 + \ldots + a_{t-1}x^{t-1}$, where a_i are some (unknown) elements of $\mathbb{Z}_p^*$ and $a(0) = k$ is a (secret) key. Each decryption teller $d_i \in \{0, 1, 2\}$ is provided with a point (x_{d_i}, y_{d_i}) on that polynomial, where x_{d_i} is publicly known and $y_{d_i} = a(x_{d_i})$ is a key share. A group of t-tellers will have to cooperate in order to reconstruct the secret k using the Lagrange interpolation formula. For the model we assume that a polynomial $a(x)$ was set and secret shares were assigned in advance.

Private variables:

- `k_share`: the teller's share of the secret;
- `x`: the value of the first variable in pair (x, y).

Procedures:

- `my_decr()`: depending on the set of participants, multiply *k_share* by a proper Lagrange basis and use the result as a key for the decryption of an input column. To determine the set of participants, a shared Boolean list *dt_paricipants* is used.

Local states:

- *idle*: waiting for the decryption phase;
- *wait*: wait for one's turn to make a decision;
- *refused*: refused to cooperate;
- *cooperating*: will participate in decryption;
- *halt*: finished his decryption.

Local transitions:

- *idle*→*wait*: enabled transition;
- *wait*→*refused*: if number of participants is already enough for key reconstruction, enable transition;
- *wait*→*cooperating*: if the number of participants is less than required for key reconstruction, enable transition; if taken, then set `dt_paricipants[dt_id]` to *true* and increment the current number of participants `dt_curr`;
- *cooperating*→*halt*: if the number of participants is sufficient, enable transition; if taken, then proceed to decryption.

3.4.6 Auditor

In order to confirm that the mix tellers performed their actions correctly, the auditor conducts an audit. In this work, we assume that the audit is based on the randomized partial checking technique, RPC in short [JJR02]. To this end, each mix teller is requested to reveal the factors for the selected half of an odd-mix batch and verify whether the input corresponds to the output. The choice to audit in-phase or after the mixing is nondeterministic. Possible selections for terms and sides of links are encoded as elements of a list.

The control flow of the Auditor module is presented in Figure 3.6.

Private variables:

- `mix_i`: index of currently audited mix teller;

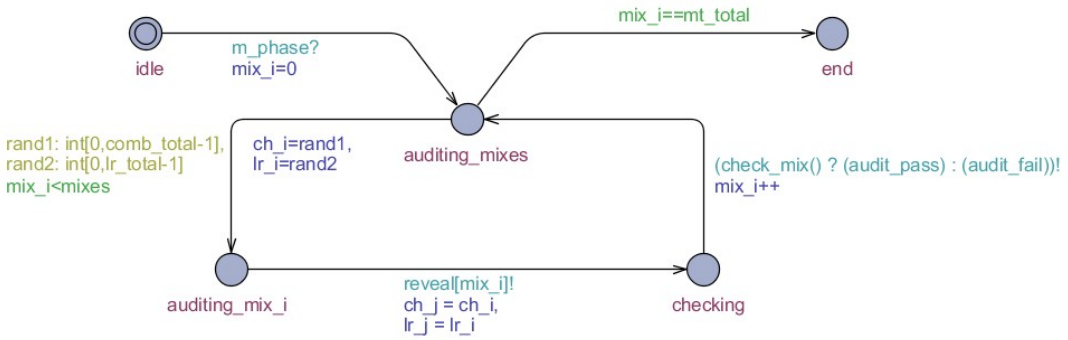


FIGURE 3.6: Auditor template.

- `ch_i`: index of `audit_ch` terms combination (a subset for audit);
- `lr_i`: index of `audit_lr` splitting, used for the left and right linkage reveal.

Procedures:

- `check_mix()`: return true if audited terms correspond to encryption of linked ones from `rev_p` using randomization factors from `rev_r`, otherwise return false.

Local states:

- *idle*: wait for the start of mixing phase;
- *auditing_mixes*: in a process of auditing mix tellers;
- *auditing_mix_i*: in a process of auditing mix teller *mix_i*;
- *cheking*: received revealed link and randomization factors from mix teller, can now proceed to correctness check;
- *end*: finished mix tellers audit.

Local transitions:

- *idle*→*auditing_mixes*: enabled transition; if taken, then set `mix_i` counter to 0;
- *auditing_mixes*→*auditing_mix_i*: if have not audited all mix tellers, enable transition; if taken, then randomly select indices `ch_i` for batch subset and `lr_i` for left-right split;
- *auditing_mix_i*→*checking*: pass the `ch_i` and `lr_i` indices to Mix Teller `mix_i` over channel `reveal[mix_i]` using shared variables `ch_j` and `lr_j`;
- *checking*→*auditing_mixes*: depending on result of `check_mix()` procedure, pass the correctness check verdict to Mix teller using either `audit_pass` or `audit_fail`, then increment `mix_i` counter by 1;
- *auditing_mixes*→*end*: if all Mix tellers were audited, enable the transition.

In the future, we plan to extend the model with auditing techniques that rely on zero-knowledge proofs.

3.4.7 Voting Infrastructure Module (Sys)

This module represents the behaviour of the election authority that prepares the ballot forms, monitors the current phase, signals the progress of the voting procedure to the other components, and at the end posts the results of the election. In addition, the module plays the role of a server that receives receipts and transfers them to the

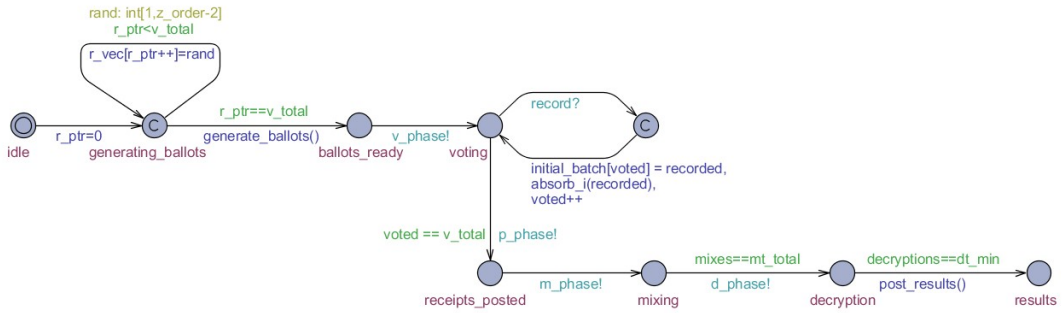


FIGURE 3.7: Module Sys.

database throughout the election. We assume that all the ballots were properly generated and thus omit procedures (e.g. ballot audits) which can ensure that. Capturing related attacks and possible defences remains a subject for future work.

Private variables:

- `vote_sum`: list of integers that maps candidates to the sum of votes they have;
- `r_vec`: list of randomization factors, used for encrypting the seed into an onion during ballot generation;
- `r_ptr`: index of `r_vec` element;
- `voted`: counter of scanned receipts.

Procedures:

- `generate_ballots()`: encrypt the list of seeds using randomization factors from `r_vec`;
- `absorb_i(recorded)`: absorb the index i of the marked cell in the `recorded` receipt into its onion and paste the result to the board;
- `post_results()` : map terms of the form g^{r+s} from the last column to the candidate x using formula $x = (r + s)\ (mod\ c\_total)$ and a look-up table `dlog`.

Local states:

- *idle*: initial state;
- *generating_ballots*: there are either no ballots or they are being generated at the moment;
- *ballots_ready*: all the ballots have been prepared;
- *voting*: election phase, when voters can obtain ballot forms and cast their votes;
- *receipts_posted*: the batch of initial receipts is now publicly seen and can be checked by the voters;
- *mixing*: wait for the re-encryption mixes to finish;
- *decryption*: wait for the decryption to finish;
- *results*: the tally is posted.

Local transitions:

- *idle*→*generating_ballots*: enabled transition; if taken, then reset `r_ptr` counter to generate a list of randomization factors;
- *generating_ballots*→*generating_ballots*: if randomization factors were not prepared, enable transition; if taken, then insert a random value `rand` into `r_vec[r_ptr]` and increment `r_ptr` iterator;

- *generating_ballots→ballots_ready*: if randomization factors are prepared, enable transition; if taken, generate ballots using randomization factors from `r_vec`;
- *ballots_ready→voting*: broadcast the start of voting phase to Voters using `v_phase` channel;
- *voting→→voting*: receive a receipt from the voter and from the committed state append it to the board, then increment a counter of receipts scanned;
- *voting→receipts_posted*: if all voters submitted receipts, enable transition; if taken, then broadcast that Voters that initial receipts were posted using the `p_phase` channel;
- *receipts_posted→mixing*: broadcast the start of a mixing phase to mix tellers using `m_phase` channel;
- *mixing→decryption*: if all mixes are done, enable transition; if taken, then broadcast the start of a decryption phase to decryption tellers using `d_phase` channel;
- *decryption→results*: if all decryptions were done, enable transition; if taken, then calculate and post results tally.

3.5 Verification and Experiments

We chose UPPAAL for this study mainly because of its modelling functionality. Interestingly, the model checking capabilities of UPPAAL turned out rather limited for analysis of voting protocols, due to the limitations of its requirement specification language. First, UPPAAL admits only a fragment of **CTL**: it excludes the "next" and "until" modalities, and does not allow for nesting of operators (with one exception that we describe below). Thus, the supported properties fall into the following categories: simple *reachability* (EFp), *liveness* (AFp), and *safety* (AGp and EGp). The only allowed nested formulas come in the form of the *p leads to q* property, written $\mathsf{p} \leadsto \mathsf{q}$, and being a shorthand for $\mathsf{AG}(\mathsf{p} \Rightarrow \mathsf{AFq})$.

Nonetheless, UPPAAL allows to model-check some simple properties of Prêt à Voter, as we show in Section 3.5.1. Moreover, by tweaking models and formulas in a clever way, one can also verify some more sophisticated requirements, see Section 3.5.2.

3.5.1 Model Checking Temporal Requirements

It is difficult to encode meaningful requirements on voting procedures in the input language of UPPAAL. We managed to come up with the following properties:

(P1) $\mathsf{EF}\,\mathsf{passed\_audit}_j$: the *j*-th mix teller might eventually pass an audit;
(P2) $\mathsf{EF}\,\mathsf{failed\_audit}_j$: the *j*-th mix teller might eventually fail an audit;
(P3) $\mathsf{AG}\,\neg\mathsf{punished}_i$: voter *i* will never be punished by the coercer;
(P4) $\mathsf{AG}\,\neg\mathsf{not\_punished}_i$: voter *i* will never escape punishment by coercer;
(P5) $\mathsf{has\_ballot}_i \leadsto \mathsf{marked\_choice}_i$: on all paths, whenever voter *i* gets a ballot form, she will eventually mark her choice.

We verified each formula on the parameterized model from Section 3.4; several configurations were used, with the number of voters ranging from 1 to 5. The verification was performed using a 32-bit version of UPPAAL 4.1.24 on a laptop with Intel i7-8665U 2.11 GHz CPU, running Ubuntu 22.04. The results are aggregated in the Table 3.1. The columns "res" and "t" denote the outcome and time in seconds respectively. The $\top$ corresponds to UPPAAL verifier returning "Property is satisfied", $\bot$ to "Property is not satisfied" accompanied with the counter-example run pasted

#Voters	⊨ P1		⊨ P2		⊨ P3		⊨ P4		⊨ P5	
	res	**t**	**res**	**t**	**res**	**t**	**res**	**t**	**res**	**t**
1	⊤	0.1	`memout`	334.4	⊥	0.1	⊥	0.1	`memout`	324.6
2	⊤	0.1	`memout`	460.4	⊥	0.1	⊥	0.1	`memout`	467.8
3	⊤	0.1	`memout`	604.6	⊥	0.1	⊥	0.1	`memout`	600.1
4	⊤	0.1	`memout`	484.2	⊥	0.1	⊥	0.1	`memout`	490.8
5	⊤	0.1	`memout`	598.7	⊥	0.1	⊥	0.1	`memout`	741.3

TABLE 3.1: Verification results for Prêt à Voter model.

into the simulator (if the diagnostic trace was selected in the option panel), and `memout` to "Out of memory".[3] The latter is caused by the state-space explosion, which is a well-known problem in the verification of distributed systems; typically, the blow-up concerns the system states to be explored in model checking and proof states in case of theorem proving.

The verification successfully terminated for the properties (P1), (P3) and (P4). This is because finding a witnessing run, which proves (P1), or a counter-example run, which violates (P3) and (P4) respectively, was sufficient without generating and exploring all possible states of the system. As opposed to the verification of (P2) and (P5), where the verifier either ran out of memory before finding a violating run, or (which is more likely) before being able to conclude that none existed after examining all the system states.[4] Therefore, it can be concluded that in the system model, there are executions where mix tellers pass an audit (P1 holds), the voters get punished (P3 does not hold) and those, where they avoid punishment (P4 does not hold).

Optimizations. To keep the model manageable and in an attempt to reduce the memory allocated per state, every numerical variable is defined as a bounded integer in the form of `int[min,max]`, restricting its range of values.[5] The states violating the bounds are discarded at run-time. For example, transition *has_ballot→marked_choice* of the Voter (Figure 3.2) has a selection of value `X` in the assignment of variable `chosen`. The type of variable `X` is `c_t`, which is an alias to `int[0,c_total-1]`, i.e., the range of meaningful candidate choices.

We also tried to keep the number of used variables minimal, as it plays an important role in the model checking procedure.

3.5.2 How to Make Model Checker Do More Than It Is Supposed To

Many important properties of voting refer to the knowledge of its participants. For example, receipt-freeness expresses that the coercer should never know how the voter has voted. Or, better still, that the coercer will never know if the voter disobeyed his instructions. Similarly, voter-verifiability says that the voter will eventually know whether her vote has been registered and tallied correctly (assuming that she follows the verification steps).

[3] Regardless of the number of voters, the verifier was running out of memory after a generation of approximately 1.2e+7 states; it is plausible to assume that the total number of reachable states could be much greater.

[4] We suspect that obtained results for (P2) and (P5) were caused by the latter, because repeating the verification process several times using randomized DFS traversal order yielded the same result.

[5] Without the explicit bounds, the range of values would be [-32768,32768] by default.

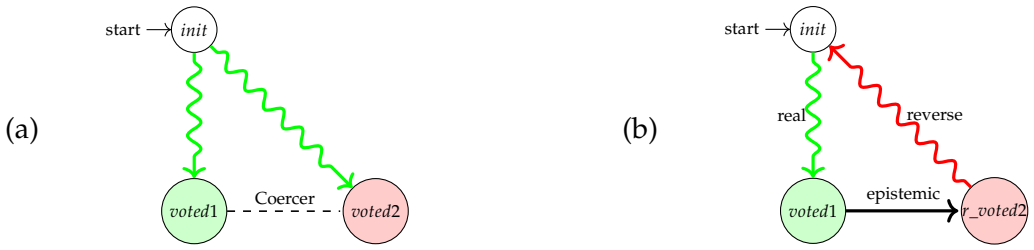


FIGURE 3.8: (a) Epistemic bisimulation triangle; (b) turning the triangle into a cycle by reversing the transition relation.

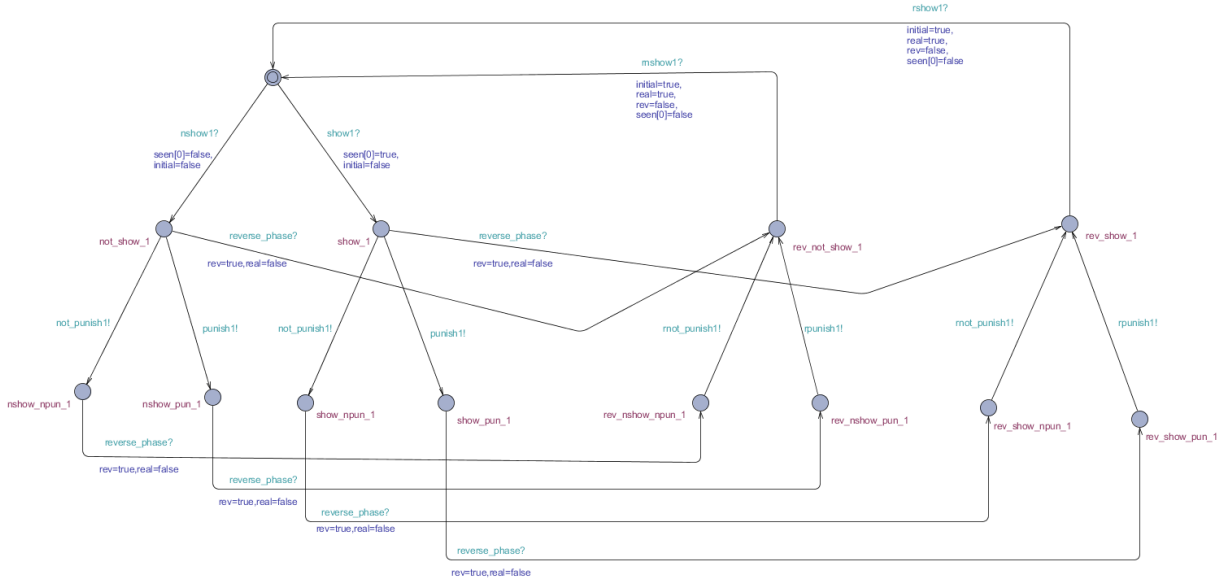


FIGURE 3.9: Coercer module augmented with the converse transition relation.

A clear disadvantage of UPPAAL is that its language for the specification of requirements is restricted to purely temporal properties. Here we show that, with some care, one can use it to embed the verification of more sophisticated properties. In particular, we show how to enable model checking of some knowledge-related requirements by a technical reconstruction of models and formulas. The construction has been inspired by the reduction of epistemic properties to temporal properties, proposed in [GJ04; Jam08]. Consequently, UPPAAL and similar tools can be used to model check some formulas of **CTLK** (i.e., **CTL** + Knowledge) that express variants of receipt-freeness and voter-verifiability.

In order to simulate the knowledge operator K_a under the **CTL** semantics, the model needs to be modified. The first step is to understand how the formula $\neg K_c \neg \mathsf{voted}_{i,j}$ (saying that the coercer doesn't know that the particular voter i hasn't voted for candidate j) is interpreted. Namely, if there is a reachable state in which $\mathsf{voted}_{i,j}$ is true, there must also exist another reachable state, which is indistinguishable from the current one, and in which $\neg\mathsf{voted}_{i,j}$ holds. The idea is shown in Figure 3.8a. We observe that to simulate the epistemic relation we need to create copies of the states in the model (the "real" states). We will refer to those copies as the *reverse states*. They are the same as the real states but with reversed transition relations. Then, we add transitions from the real states to their corresponding reverse states, that simulate

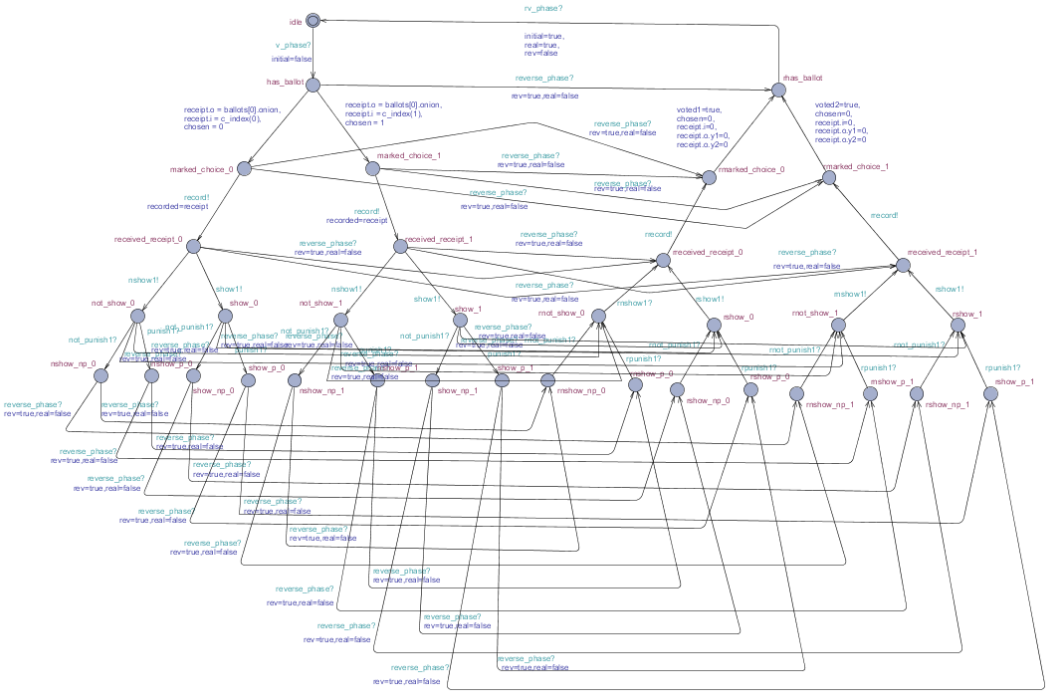


FIGURE 3.10: Voter1 with reversed transitions.

the epistemic relation between the states. This is shown in Figure 3.8b.

To illustrate how the reconstruction of the model works on a concrete example, we depict the augmented templates in Figures 3.9–3.14.

In order to effectively modify the model and verify the selected properties according to the previously defined procedure, the model was first simplified. In the simplified version there are two voters and the coercer can interact only with one of them. Furthermore, we removed the verification phase and the tallying phase from the model.

The next step is the reconstruction of formulas. Let us take the formula for the weak variant of receipt-freeness from Section 3.2.2, i.e., $\mathsf{EF}(\mathsf{results} \wedge \neg \mathsf{voted}_{i,j} \wedge \neg K_c \neg \mathsf{voted}_{i,j})$. In order to verify the formula in UPPAAL, we need to replace the knowledge operator according to our model reconstruction method (see Figure 3.8 again). This means that the verifier should find a path that closes the cycle: from the initial state, going through the real states of the voting procedure to the vote publication phase, and then back to the initial state through the reversed states. In order to "remember" the relevant facts along the path, we use persistent Boolean variables $\mathsf{voted}_{i,j}$ and $\mathsf{negvoted}_{i,j}$: once set to true they always remain true. We also introduce a new persistent variable $\mathsf{epist\_voted}_{i,j}$ to refer to the value of the vote after an epistemic transition. Once we have all that, we can propose the reconstructed formula: $\mathsf{EF}(\mathsf{results} \wedge \mathsf{negvoted}_{i,j} \wedge \mathsf{epist\_voted}_{i,j} \wedge \mathsf{initial})$. UPPAAL reports that the formula holds in the model.

A stronger variant of receipt-freeness is expressed by another formula — already seen in Section 3.2.2 — $\mathsf{AG}(\mathsf{results} \Rightarrow \neg K_c \neg \mathsf{voted}_{i,j})$. Again, the formula needs to be rewritten to a pure **CTL** formula. As before, the model checker should find a cycle from the initial state, "scoring" the relevant propositions on the way. More precisely, it needs to check if, for every real state in which the election has ended, there exists a path going back to the initial state through a reverse state in which the

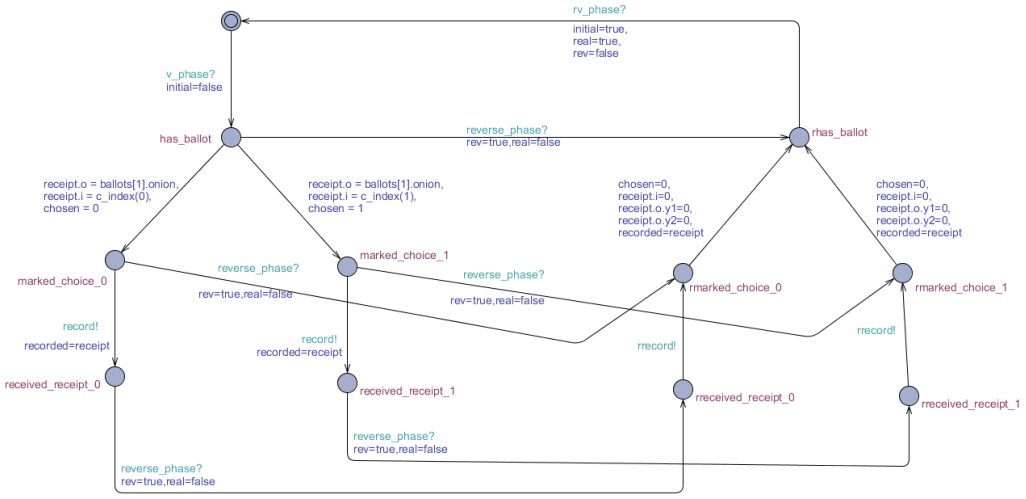


FIGURE 3.11: Voter2 with reversed transitions.

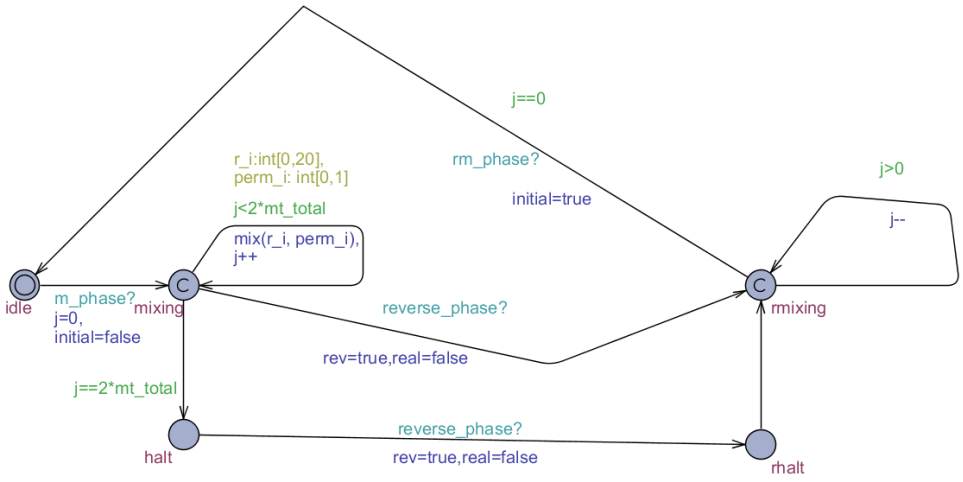


FIGURE 3.12: Mix Teller with reversed transitions.

voter has voted for the selected candidate. This can be captured by the following formula: $\mathsf{AG}\big((\mathsf{results} \wedge \mathsf{real}) \Rightarrow \mathsf{EF}(\mathsf{voted}_{i,j} \wedge \mathsf{init})\big)$. Unfortunately, this formula cannot be verified in UPPAAL, as UPPAAL does not allow for nested path quantifiers. In the future, we plan to run the verification of this formula using another model checker LTSmin [Kan+15] that accepts UPPAAL models as input but allows for more expressive requirement specifications.

3.6 Replicating Pfitzmann's Attack

A version of *Pfitzmann's attack* is known to compromise mix-nets with randomized partial checking [KW13]. It can be used to break the privacy of a given vote with probability 1/2 of being undetected during RPC. The leaked information may differ depending on both the implementation of the attack and the voting protocol.

The idea is that the first mix teller, who is corrupted, targets a ciphertext c_i from the odd mix input, and replaces some output term c_j with c_i^{δ}. After the results of

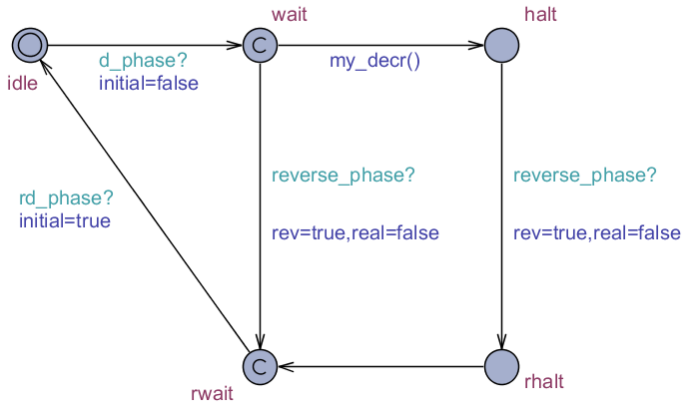


FIGURE 3.13: Decryption Teller with reversed transitions.

#Voters	⊨ P1		⊨ P2$^{(j=0)}$		⊨ P3		⊨ P4		⊨ P5	
	res	**t**	**res**	**t**	**res**	**t**	**res**	**t**	**res**	**t**
2	⊤	0.1	⊤	0.1	⊥	0.1	⊥	0.1	memout	424.7
3	⊤	0.1	⊤	0.1	⊥	0.1	⊥	0.1	memout	628.2
4	⊤	0.1	⊤	0.1	⊥	0.1	⊥	0.1	memout	569.4
5	⊤	0.1	⊤	0.1	⊥	0.1	⊥	0.1	memout	628.4

TABLE 3.2: Verification results for Prêt à Voter model, where the first mix teller is corrupt.

decryption are posted, a pair of plaintext messages m and m' satisfying the equation $m' = m^{\delta}$ can be used to identify the corresponding input terms.

Clearly, the model presented in Section 3.4 is too basic to allow for the detection of the attack. Instead, we can examine the attacker's behaviour by a simple extension of the model. For that, we change the Mteller template as shown in Figure 3.15. The only difference lies in how the first re-encryption mix is done: the corrupted mix teller targets c_0, chooses a random non-zero δ, and uses c_0^{δ} instead of some other output term. We assume that the corrupt mix teller will always try to cheat. In all other respects, the teller behaves honestly.

Using UPPAAL, it can be verified that there are executions where the corrupt mix teller's cheating behaviour is not detected during the audit. That is, both (P2) EF $\mathsf{failed\_audit}_0$ and (P1) EF $\mathsf{passed\_audit}_0$ produce "Property satisfied" as the output. The verification results are reported in Table 3.2, omitting the configuration with 1 Voter, where the behaviour of a corrupt mix teller is identical to the honest one. Similarly to the results from Section 3.5, verification successfully terminates finding a run that proves (P1), and runs that violate (P3) and (P4), and fails in the case of (P5) due to a lack of memory. However, in contrast to the results obtained with honest Mix tellers, here property (P2) ends up being satisfied for $j = 0$ (i.e., for the corrupt Mix teller). Therefore, in this modified scenario, where the first mix teller is corrupt, the audit may sometimes detect the cheating (P2 holds) as well as fail to notice it (P1 holds); the conclusions for (P3) and (P4) remain the same.

We note that, in order to successfully verify those properties in our model of Prêt à Voter, the search order option in UPPAAL had to be changed from the (default) `Breadth First` to either `Depth First` or `Random Depth First`.

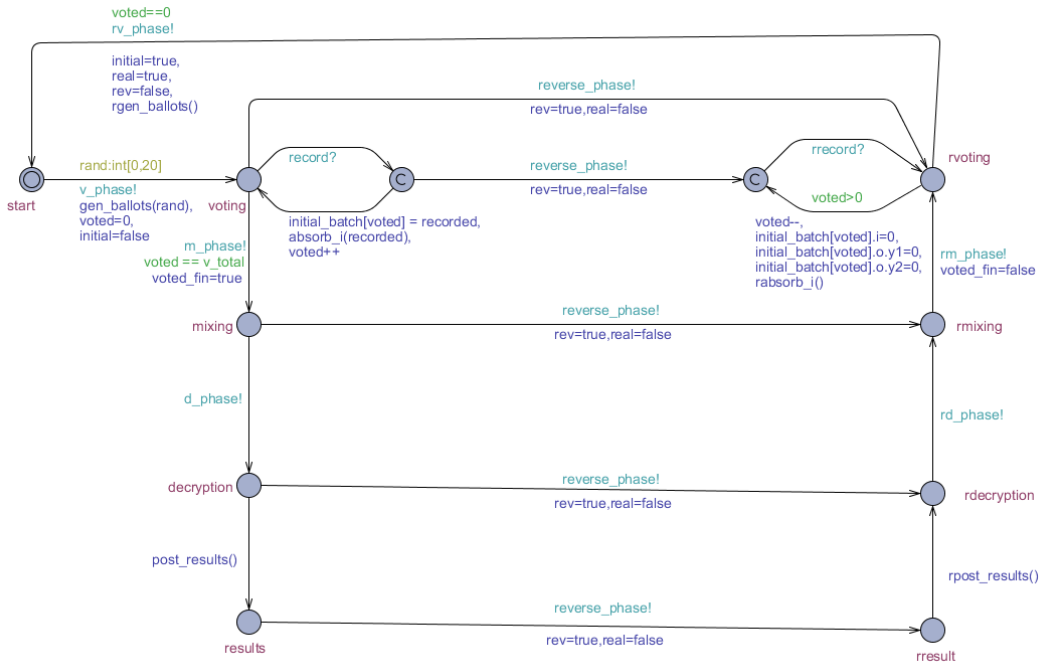


FIGURE 3.14: Sys module with reversed transitions.

3.7 Related Work

Coercion-resistant and voter-verifiable voting systems. Over the years, the properties of *ballot secrecy*, *receipt-freeness*, *coercion resistance*, and *voter-verifiability* were recognized as important for an election to work properly. In particular, receipt-freeness and coercion-resistance were studied and formalized in multiple ways [BT94; Oka98; DKR06; KTV10a; DLL12], see also [Men09; TJR16] for an overview. More recently, significant progress has been made in the development of voting systems that would be coercion-resistant and at the same time allow the voter to verify "her" part of the election outcome [RST15; Cor+16]. A number of secure and voter-verifiable schemes have been proposed, notably Prêt à Voter for supervised elections [CRS05; Rya10], Pretty Good Democracy for internet voting [RT13], and Selene, a coercion mitigating form of tracking number-based, internet scheme [RRI16].

Such schemes are starting to move out of the laboratory and into use in real elections. For example, (a variant of) Prêt à Voter has been successfully used in one of the state elections in Australia [Bur+12] while the Scantegrity II system [Cha+09] was used in municipal elections in the Takoma Park county, Maryland. Moreover, a number of verifiable schemes were used in non-political elections. E.g., Helios [Adi08] was used to elect officials of the International Association of Cryptologic Research and the Dean of the University of Louvain la Neuve. This underlines the need for extensive analysis and validation of such systems.

Formal verification of voting protocols. In voting systems, *verifiable* means that the voters are able to verify the outcome of the election. This is different from *formal verification* whose task is to provide algorithmic tools for checking if a system satisfies a given requirement. General verification tools for security protocols include ProVerif [CB13], Scyther [CM12], AVISPA [Arm+05], and Tamarin [Mei+13].

Formal analysis of selected voting protocols, based on theorem proving in first-order logic or linear logic, includes attempts at verification of vote counting in [BGS13;

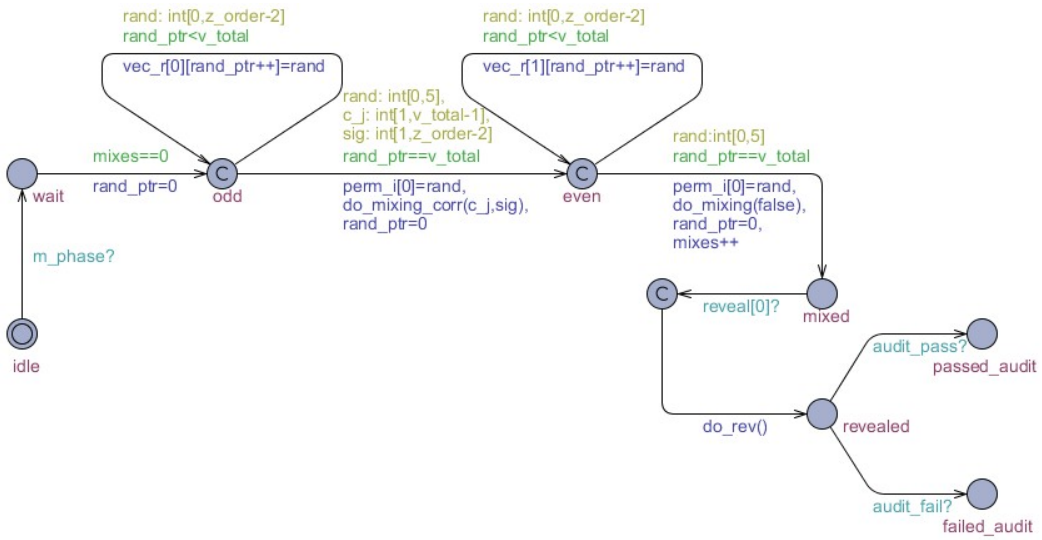


FIGURE 3.15: Corrupted Mix Teller module.

PS15]. The Coq theorem prover for higher-order logic [Ber+04] was used to implement the STV counting scheme in a provably correct way [Gha+18], and to produce a provably voter-verifiable variant of the Helios protocol [HGT19]. Moreover, Tamarin [Mei+13] was used to verify receipt-freeness in Selene [BDS17] and Electryo [ZRR20]. Approaches based on model checking are fewer and include the analysis of risk-limiting audits [Bec+16] with the CBMC model checker [CKL04].

Multi-agent models and model checkers in verification of security. Multi-agent model checking is virtually unexplored in analysis of voting systems. The only relevant work that we are aware of is [JKK18] where a simple multi-agent model of Selene was proposed and verified using the MCMAS model checker [LQR17]. Related research includes the use of multi-agent methodologies to specify and verify properties of authentication and key-establishment protocols [LP08; BKL16] with MCMAS. In particular, [BKL16] used MCMAS to obtain and verify models, automatically synthesized from high-level protocol description languages such as CAPSL, thus creating a bridge between multi-agent and process-based methods.

In all the above cases, the focus is on the verification itself. Indeed, all the tools mentioned above provide only a text-based interface for the specification of the system. As a result, their model specifications closely resemble programming code, and insufficiently protect from the usual pitfalls of programming: unreadability of the code, lack of modularity, and opaque control structure. In this work, we draw attention to tools that promote modular design of the model, emphasize its control structure, and facilitate inspection and validation.

3.8 Conclusions

Formal methods are well established in proving (and disproving) the correctness of cryptographic protocols. What makes voting protocols special is that they prominently feature human and social aspects. In consequence, an accurate specification of the behaviours admitted by the protocol is far from straightforward. An environment that supports the creation of modular, compact, and – most of all – readable specifications can be an invaluable help in the design and validation of voting systems.

In this context, the UPPAAL model checker has a number of advantages. Its modelling language encourages modular specification of the system behaviour. It provides flexible data structures and allows for parameterized specification of states and transitions. Last but not least, it has a user-friendly GUI. Clearly, a good graphical model helps to understand how the voting procedure works and allows for preliminary validation of the system specification just by looking at the graphs. Anybody who ever inspected a text-based system specification or the programming code itself will know what we mean.

In this work, we try to demonstrate the advantages of UPPAAL through a case study based on a version of Prêt à Voter. The models that we have obtained are neat, easy to read, and easy to modify. On the other hand, UPPAAL has not performed well with the verification itself. This was largely due to the fact that its requirement specification language turned out to be very limited – much more than it seemed at first glance. We managed to partly overcome the limitations by a smart reconstruction of models and formulas. In the long run, however, a more promising path is to extend the implementation of verification algorithms in UPPAAL so that they handle nested path quantifiers and knowledge modalities, given explicitly in the formula.

The model proposed here is far from complete. We intend to refine and expand it to capture a broader range of attacks, in particular coercion (or vote-buying attacks) that involve subtle interactions between coercer and voters. Prime examples include chain voting and randomization attacks, where the coercer requires the voter to place an "X" in, say, the first position. Such an attack does not violate any privacy property – the coercer does not learn the vote – but it does deny the voter the freedom to cast her vote as intended. Still, more subtle styles of attack have been identified against many verifiable schemes by Kelsey, [Kel+10]. Essentially any freedom the voter may have in executing the voting ceremony can potentially be exploited by a coercer.

A comprehensive discussion of coercion-resistance and its possible formalizations is also planned for future work. Another important line of research concerns data independence and saturation results. It is known that to verify some properties, it suffices to look for small counterexamples [ACK16]. It is also known that such results are in general impossible [GS92] or incur prohibitive blowup [Cze+19]. We will investigate what saturation can be achieved for the verification of Prêt à Voter.

Chapter 4

Practical Abstraction for Model Checking of Multi-Agent Systems

The experiments in Chapter 3 have shown that formal verification of voting procedures faces a substantial complexity barrier. This coincides with the fact that model checking of multi-agent systems (MAS) is known to be hard, both theoretically and in practice. The state-space explosion is a major challenge here, as faithful models of real-world systems are immensely huge and infeasible even to generate – let alone

verify them. A smart abstraction of the state space may significantly reduce the model, and facilitate the verification. However, while state abstraction is well studied from the theoretical point of view, little work has been done on how to define actual abstractions in practice. 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 the 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 and a scenario of gossip learning for social AI.

4.1 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 [Woo02; SL09]. In particular, the temporal logics **CTL**, **LTL**, and **CTL*** provide formalizations of many relevant properties, including reachability, liveness, safety, and fairness [Eme90]. Algorithms and tools for verification of such properties have been in constant development for 40 years, with temporal model checking being the most popular approach [BK08; Cla+18].

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 [Sch03; BDJ10]. No less importantly, it is often unclear how to create the input model, especially if the system to be modelled involves human behaviour [Jam+20b]. Similarly, formalizing the relevant properties in the right way is by no means trivial [Jam+21]. Both parts of the specification are error-prone and difficult to debug and validate, and most model-checkers for MAS do not even have a graphical user interface.[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 [McM93; McM02; PL03; KLP04; LP07; Hv14; LQR17] fall into the former group. Model reduction methods include partial-order reduction [Pel93; Ger+99; Jam+20a], equivalence-based reductions [de +84; Alu+98; Bel+21], and state abstraction [CC77], see Section 4.7 for a detailed discussion of the related work.

Towards practical abstraction. A smart abstraction of the state space may reduce the model to a 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 [Cla+00b] 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

[1] Notable exceptions include UPPAAL [BDL04] and STV [Kur+21].

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 work, we suggest that the conceptually simplest kind of abstraction consists in removing a domain 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 **CTL*** (**ACTL***). More precisely, our *may*-abstractions preserve the falsity of **ACTL*** properties, so if $\varphi \in$ **ACTL*** holds in the abstract model, it must also hold in the original one. Conversely, our *must*-abstractions preserve the truth of **ACTL*** formulae, so if $\varphi \in$ **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.

4.2 Preliminaries

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

4.2.1 MAS Graphs

To represent the behaviour of a multi-agent system, we use modular representations inspired by reactive modules [AH99], interleaved interpreted systems [LPQ10; Jam+20a], and in particular by the way distributed systems are modelled in UPPAAL [BDL04].

Let *Var* be a finite set of typed variables over finite domains.[2] By $Eval(Var)$ we denote a set of evaluations, i.e., functions mapping variables $v \in Var$ to the values from their domains $dom(v)$. From variables in *Var* and constants in $\bigcup_{v \in Var} dom(v)$ the expressions can be composed using arithmetic operators (i.e., "$+$", "$-$", "$*$", "$/$" and "%") in the usual way, e.g., $2x + 1$. Similarly, the atomic logical conditions are constructed from expressions using relation symbols (i.e., "$\leq$", "$<$", "$\neq$", "$=$", "$>$", "$\geq$"), e.g., $2x + 1 > 0$. They can be further combined with logical connectives (i.e., "$\neg$", "$\wedge$", "$\vee$") to form compound logical conditions. The set of all possible logical conditions (also called *guards*) over *Var* is denoted by *Cond*. Furthermore, any $g \in Cond$ can be associated with its set of satisfying evaluations $Sat(g) = \{\eta \in Eval(Var) \mid \eta \models g\}$.

[2] We consider only variables with finite domains, in line with most model checking algorithms and tools for MAS.

Definition 4.1 (Agent graph). *An **agent graph** , describing the behaviour of an agent, is a tuple $G = (Var, Loc, l_0, g_0, Act, Effect, Chan, \hookrightarrow)$, consisting of:*

- *Var: a finite set of typed **variables** over finite domains;*
- *Loc: a non-empty finite set of **locations**;*
- *$l_0 \in Loc$: the initial location;*
- *$g_0 \in Cond$: the initial condition, required to be satisfied at the initial location; For simplicity, we assume that $Sat(g_0) = \{\eta_0\}$, i.e. there is an initial value $v_0 = \eta_0(v)$ for every $v \in Var$;*
- *Act: a set of **actions**, with $\tau \in Act$ standing for "do nothing";*
- *$Effect : Act \times Eval(Var) \mapsto Eval(Var)$: the **effect** of an action; We assume $Effect(\tau, \eta) = \eta$ for any $\eta \in Eval(Var)$;*
- *Chan: a finite set of asymmetric one-to-one synchronization **channels**; Furthermore, we define the set of synchronizations as $Sync = \{c!, c? \mid c \in Chan\} \cup \{-\}$, with $c!$ and $c?$ for sending and receiving on a channel c, respectively, and "$-$" for no synchronization;*
- *$\hookrightarrow \subseteq Loc \times Label \times Loc$: a set of **labelled edges** with labels from $Label \subseteq Cond \times Sync \times Act$, which will be used to define the local transition relation.*

Instead of $(l, labl, l') \in \hookrightarrow$, we will often write $l \xhookrightarrow{g:ch\,\alpha} l'$, where $g = cond(labl)$, $ch = sync(labl)$ and $\alpha = act(labl)$. The *cond*, *sync*, *act* are destructors mapping a label to its counterpart. We will often omit writing $g = \top$, $ch = -$ or $\alpha = \tau$ explicitly within a label.

An edge labelled by $labl \in Label$ is said to be *locally enabled* for evaluation $\eta \in Eval(Var)$ iff $\eta \models cond(labl)$.

Furthermore, every action $\alpha \in Act \setminus \{\tau\}$ can be associated with a non-empty finite sequence of atomic assignments (also called *updates*) of the form $\alpha^{(1)}\alpha^{(2)} \ldots \alpha^{(m)}$.[3] It shall be assumed that such atomic sequences are initially *normalized*, in the sense that a variable can appear on the left-hand side of an assignment statement at most once.[4]

Function $\mathcal{V} : Cond \cup Act \mapsto \mathcal{P}(Var)$ returns the set of variables occurring in the associated expression(s).

Without loss of generality, we assume that the variables in $Var = \{v_1, \ldots, v_k\}$ are ordered in an arbitrary way. Thus, an evaluation of $V \subseteq Var$, where $V = \{v_{i_1}, \ldots, v_{i_l}\}$, $l \leq k$ and $i_j \in \{1, \ldots, k\}$, can be seen as a vector $\eta(V) = [\eta(v_{i_1}), \ldots, \eta(v_{i_l})]$, where $i_j < i_{j+1}$ for $1 \leq j \leq l-1$. Moreover, we say that a pair of evaluations $\eta_1 \in Eval(Var_1)$ and $\eta_2 \in Eval(Var_2)$ *agrees on* the variables $V \subseteq Var_1 \cap Var_2$ (denoted $\eta_1 =_V \eta_2$) iff $\eta_1(V) = \eta_2(V)$, i.e., $\eta_1(v) = \eta_2(v)$ for every $v \in V$.

Let $V \subseteq Var$, $c \in dom(V)$, $\eta \in Eval(Var)$, $g \in Cond$, $\alpha \in Act$. By $\eta[V = c]$ and $g[V = c]$ we denote the result of (straightforward) substitution of occurring variables from V with corresponding element(s) of c. The $\alpha[V = c]$ corresponds to a more nuanced substitution, or rather a chain of substitutions over the sequence of atomic updates $\alpha^{(i)}$. This is because sequential updates $\alpha^{(i)}$ might be internally dependent (i.e., when the variable from the left-hand side of some $\alpha^{(j)}$ appears as parameter later on the right-hand side of some $\alpha^{(i)}$, where $j < i$). Therefore, the $\alpha[V = c]$ would correspond to $\alpha^{(m)}[V^{(m)} = c^{(m)}] \circ \ldots \circ \alpha^{(1)}[V^{(1)} = c^{(1)}]$, where $V^{(1)} = V \cap \mathcal{V}(\alpha^{(1)})$, $V^{(i)} = V \cap \mathcal{V}(\alpha^{(i)}) \setminus \{lhs(\alpha^{(j)}) \mid j < i\}$ and $c^{(i)} = c|_{V^{(i)}}$.

Note that for any pair of $g \in Cond$ and $\eta \in Eval(Var)$ it holds that: $\eta \models g$ iff $g[Var = \eta(Var)]$ evaluates to $\top$.

[3] We will often write such sequences in a form of composition $\alpha^{(m)} \circ \ldots \circ \alpha^{(2)} \circ \alpha^{(1)}$, where "$\circ$" is a right-associative operator.

[4] Note that this is a purely syntactic condition that is easy to check and does not constrain the expressive power (e.g., one can always introduce auxiliary variables).

Remark 4.1. *For an action α, there can be more than one (normalized) sequence of atomic updates representing it. For example, when some underlying $\alpha^{(i)}$ and $\alpha^{(i+1)}$ are commutative. W.l.o.g., in the sequel we assume that any pair of updates (or their atomic sequences) that yields the same result are equal, i.e., $\alpha_1 = \alpha_2$ iff $\forall_{\eta \in Eval(Var)} Effect(\alpha_1, \eta) = Effect(\alpha_2, \eta)$.*

Furthermore, note that effect of α can only change the valuation for the variables appearing on the left-hand side of its atomic updates $L = \bigcup_{i=1}^{m} lhs(\alpha^{(i)})$, that is for any $\eta, \eta' \in Eval(Var)$ the following holds:

$$Effect(\alpha, \eta) = \eta' \qquad implies \qquad \eta =_{(Var \setminus L)} \eta'$$

Moreover, the outcome of $Effect(\alpha, \eta)$ depends only on the evaluation of the variables appearing on the right-hand side of its atomic updates that did not appear on the left-hand side of preceding elements of the underlying atomic sequence $R = \bigcup_{i=1}^{m} \left(rhs(\alpha^{(i)}) \setminus (\bigcup_{j=1}^{i} lhs(\alpha^{(j)})) \right)$, that is for any $\eta, \eta' \in Eval(Var)$ the following holds:

$$Effect(\alpha, \eta) =_L Effect(\alpha[R = \eta(R)], \eta')$$

Definition 4.2 (MAS graph). *A **MAS graph** is a multiset[5] of agent graphs with a distinguished set of shared (also called global) variables; in order to avoid ambiguity in the notation, the set of shared variables will be separated from the agent graphs by semicolon. We assume, w.l.o.g., that all local variables have unique names.[6] Then, the set of shared variables can be seen as those that occur in at least two different agent graphs.*

FIGURE 4.1: MAS graph for ASV: G^{Voter} (left) and $G^{Coercer}$ (right).

Example 4.1 (ASV). *As the running example, we use a variation of the Asynchronous Simple Voting scenario of [Jam+20a], where the number of candidates is fixed to 3. Its MAS graph $ASV = \{|Var_{sh}; G^{Voter}, G^{Coercer}|\}$ is shown in Figure 4.1.*

The G^{Voter} has locations $Loc^{Voter} = \{\texttt{idle}, \texttt{voted}, \texttt{disobeyed}, \texttt{obeyed}\}$, where `idle` is the initial location, and variables $Var^{Voter} = \{\texttt{vt}, \texttt{sh}\}$, where `vt` represents the chosen candidate and `sh` is used for the possible communication with the coercer over the set of synchronization channels $Chan^{Voter} = Chan^{Coercer} = \{g, ng\}$. The $G^{Coercer}$ has locations $Loc^{Coercer} = \{\texttt{idle}, \texttt{halt}\}$, where `idle` is the initial location, and variables $Var^{Coercer} = \{\texttt{Kref}, \texttt{Kvt}, \texttt{sh}\}$, where `Kref` and `Kvt` represent the awareness of voter's refusal to obey and shown voting receipt respectively. The variables `vt`, `sh` and `Kvt` are of the bounded integer type with domain of values $\{0, 1, 2, 3\}$, and `Kref` is a Boolean variable with domain of

[5] We denote a multiset container with "{|" and "|}" brackets to avoid confusion with ordinary sets.

[6] This can be ensured, e.g., by prefixing the identifiers of local variables with the name of the agent graph; in case there are multiple agent graphs of the same type, some enumeration over agent graphs of the same type can be established, so that their index is included into the auxiliary prefix.

values $\{0,1\}$. *The initial locations are marked by double-circle, and the initial condition is* $g_0 \equiv \bigwedge_{v \in Var}(v{=}0)$, *where* $Var = Var^{Voter} \cup Var^{Coercer}$.

The voter — represented by the agent graph G^{Voter} *— starts by nondeterministically selecting one of the three candidates, for whom the vote will be cast (*`idle`→`voted`*). 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 coercer — represented by the agent graph* $G^{Coercer}$. *In turn, the coercer either gets the proof and learns for whom the vote was cast, or becomes aware of the voter's refusal.*

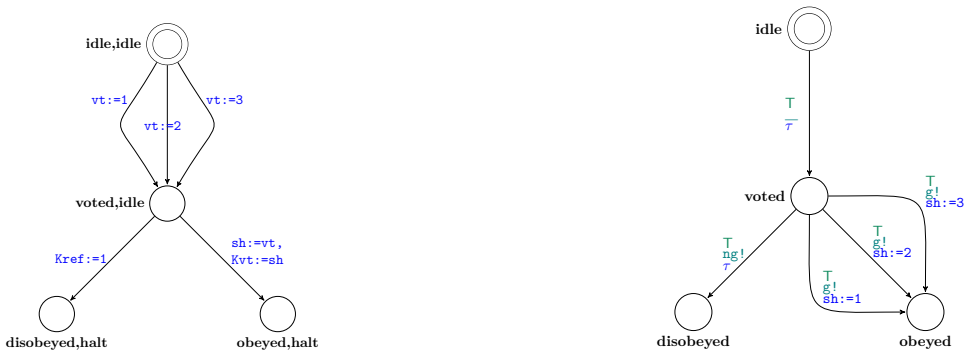


FIGURE 4.2: (a) Combined MAS graph of ASV. (b) *May*-abstraction $\mathcal{A}^{may}_{\{x\}}(G^{Voter}, ASV)$.

4.2.2 Models of MAS Graphs

We define the execution semantics of a MAS graph by its *unwrapping*.

Definition 4.3 (Combined MAS graph). *Let* $MG = \{|Var_{sh}; G^1, \ldots, G^n|\}$ *be a MAS graph having a set of shared variables* Var_{sh}. *The* ***combined MAS graph*** *of MG is the agent graph* $G_{MG} = (Var, Loc, l_0, g_0, Act, Effect, Chan, \hookrightarrow)$, *where* $Var = \bigcup_{i=1}^{n} Var_i$, $Loc = Loc^1 \times \ldots \times Loc^n$, $l_0 = (l_0^1, \ldots, l_0^n)$, $g_0 = g_0^1 \wedge \ldots \wedge g_0^n$, $Act = \bigcup_{i=1}^{n} Act^i$, $Chan = \varnothing$.

Relation $\hookrightarrow$ *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* $1 \leq i, j \leq n$*):*

$$\frac{l_i \xrightarrow{g_i : c!\alpha_i}_i l_i' \wedge l_j \xrightarrow{g_j : c?\alpha_j}_j l_j'}{(l_i, l_j) \xrightarrow{g_i \wedge g_j : (\alpha_j \circ \alpha_i)} (l_i', l_j')} \qquad \frac{l_i \xrightarrow{g_i : \alpha_i}_i l_i'}{(l_i, l_j) \xrightarrow{g_i : \alpha_i} (l_i', l_j)}$$

$$\frac{l_i \xrightarrow{g_i : c?\alpha_i}_i l_i' \wedge l_j \xrightarrow{g_j : c!\alpha_j}_j l_j'}{(l_i, l_j) \xrightarrow{g_i \wedge g_j : (\alpha_i \circ \alpha_j)} (l_i', l_j')} \qquad \frac{l_j \xrightarrow{g_j : \alpha_j}_j l_j'}{(l_i, l_j) \xrightarrow{g_j : \alpha_j} (l_i, l_j')}$$

Lastly, the effect function is defined by:

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

Remark 4.2. *In the definition above, the function* $Effect^i : Act^i \times Eval(Var) \mapsto Eval(Var)$ *is a natural extension of the original one from agent graph* G^i*, s.t.* $Var^i \subseteq Var$ *and* $\Big(Effect^i(\alpha, \eta) = \eta'\Big) \Rightarrow \Big(\forall v \in (Var \setminus Var^i) \,.\, \eta =_v \eta'\Big)$.[7]

Example 4.2. *The combined MAS graph* G_{ASV} *for asynchronous simple voting of Example 4.1 is depicted in Figure 4.2a.*

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 $labl \in Label$, s.t. $sync(labl) = -$. To turn it into a model, we still need to instantiate the variables in combined MAS graph with their possible values.

The nodes and edges in an agent graph G correspond to *sets* of states and transitions, defined by the unwrapping of G.

Definition 4.4 (Unwrapping)**.** *Let* $G = (Var, Loc, l_0, g_0, Act, Effect, Chan, \hookrightarrow)$ *be an agent graph. The* ***unwrapping*** *of* G *over* $AP \subseteq Loc \cup Cond$ *is a 5-tuple* $\mathcal{M}(G) = (St, I, \longrightarrow, AP, L)$*, where:*

- $St = Loc \times Eval(Var)$,
- $I = \{\langle l_0, \eta\rangle \in St \mid \eta \in Sat(g_0)\}$,
- $\longrightarrow = \longrightarrow_0 \cup \{(s, s) \in St \times St \mid \neg\exists s' \in St \,.\, s \longrightarrow_0 s'\}$, *where* $\longrightarrow_0 = \{(\langle l, \eta\rangle, \langle l', \eta'\rangle) \in St \times St \mid \exists\, l \overset{g:\alpha}{\hookrightarrow} l' \,.\, \eta \in Sat(g) \wedge \eta' = Effect(\alpha, \eta)\}$,[8]
- $AP \subseteq Loc \cup Cond$,
- $L : St \to 2^{AP}$, *such that* $L(\langle l, \eta\rangle) = \{l\} \cup \{g \in Cond \mid \eta \in Sat(g)\}$.

We call $\mathcal{M}(G)$ *the* ***model*** *of an agent graph* G*. For a MAS graph* MG *the model* $\mathcal{M}(MG)$ *is given by the unwrapping of its combined graph, i.e.* $\mathcal{M}(MG) = \mathcal{M}(G_{MG})$.

Note that a model $\mathcal{M}(G)$ *(resp.* $\mathcal{M}(MG)$*) over* AP *is finite iff* AP *is finite.*

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 allow 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 4.3. *The unwrapping of the MAS graph for asynchronous simple voting (over any AP) is shown in Figure 4.3, with the initial state* $\langle$`idle,idle,vt=0,sh=0,Kref=0,Kvt=0`$\rangle$.

Remark 4.3. *MAS graph* $MG = \{|Var_{sh}; G^1, \ldots, G^n|\}$ *could also be "unwrapped" into extended model (see Definition 2.21), where indistinguishability relation* $\sim_i$ *(for* $i = 1 \ldots n$*) is given by:*

$$\langle (l^1, \ldots, l^n), \eta\rangle \sim_i \langle (\hat{l}^1, \ldots, \hat{l}^n), \hat{\eta}\rangle \quad \textit{iff} \quad l^i = \hat{l}^i \wedge \eta =_{Var_i} \hat{\eta}$$

Lemma 4.1. *For any agent graph, the number of states in its unwrapping is at most*

$$|Loc| \cdot \prod_{v \in Var} |dom(v)|.$$

Proof. Follows from the definition $St \subseteq Loc \times Eval(Var)$ and the fact that both Loc and $Eval(Var)$ are finite (in particular, Var is finite and $|dom(v)| < \infty$ for every $v \in Var$). □

[7] In other words, the (extended) effect exhibits the original behaviour w.r.t. the (original) variables Var^i of the agent i, and leaves the evaluation for all other variables unaffected.

[8] We add loops wherever necessary to make the relation serial.

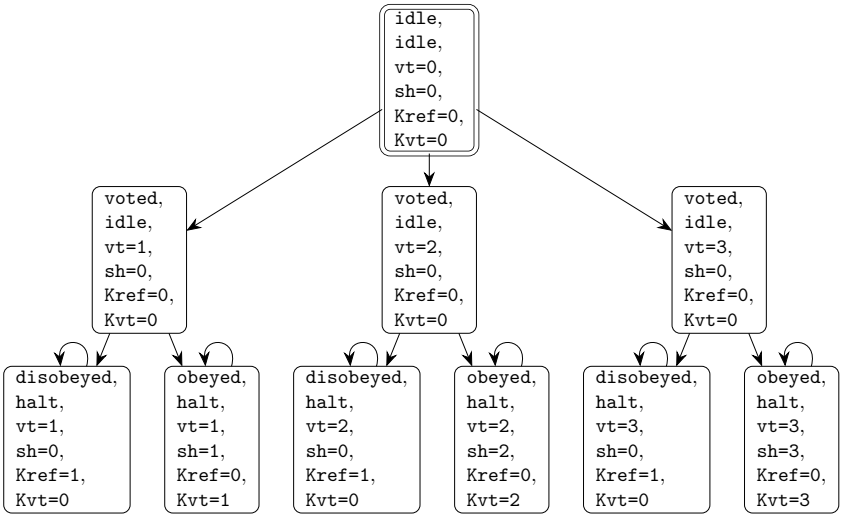


FIGURE 4.3: Unwrapping $\mathcal{M}(ASV)$ for Asynchronous Simple Voting.

It is easy to construct an example where the bound is tight. Thus, in the worst case, the number of states in the unwrapping of agent graph grows *exponentially* in the number of variables.

Definition 4.5 (Local Domain)**.** *A **local domain** is a function $d : Loc \mapsto \mathcal{P}(Eval(Var))$ that maps each location $l \in Loc$ to the set of its reachable evaluations (i.e., for which there exists a corresponding reachable state $\langle l, \eta \rangle$ in the model). By $d(l)|_V = \{\eta|_V \mid \eta \in d(l)\}$ we denote the restriction of evaluations in $d(l)$ that considers only the variables $V \subseteq Var$.*

4.2.3 Branching-Time Logic ACTL*

To specify requirements, we use the *universal fragment of the branching-time logic* **CTL*** [Eme90], denoted **ACTL***[9] with A ("for every path") as the only path quantifier. We refer to Definition 2.10 and Definition 2.11 for details on its syntax and semantics.

Example 4.4. *Consider the unwrapping $\mathcal{M}(ASV)$ of ASV depicted in Figure 4.3 over $AP = \{$`Kvt=vt`, `Kref=1`, `Kvt>0`, `obeyed`, `disobeyed`$\}$, for which the mappings of the function L can be derived intuitively.*

The model $M = \mathcal{M}(ASV)$ satisfies the **ACTL*** *formula* AG($\neg$`obeyed` $\vee$ `Kvt=vt`)*, saying that if Voter obeys, Coercer gets to know how she voted, and the formula* AG($\neg$`disobeyed` $\vee$ `Kref=1`)*, expressing that she cannot disobey Coercer's instructions without his knowledge. It does not however satisfy* AF(`Kvt>0`)*, saying that Coercer will eventually get to know how the Voter voted.*

4.3 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 [DGG97; GJ02]. Typically, they take concrete states and cluster them according to a given equivalence relation. The *may* model includes transitions of type $\exists\exists$, i.e., $[s_1] \longrightarrow [s_2]$ in the abstract model iff $\exists_{s_1' \in [s_1]} \exists_{s_2' \in [s_2]} s_1' \longrightarrow s_2'$ in the concrete model. The *must* model includes transitions of type $\forall\exists$, i.e., $[s_1] \longrightarrow [s_2]$ in the

[9] Not to be confused with "Action **CTL**" of [NV90].

Algorithm 1: Abstraction of MAS graph MG wrt V

```
for MG = {|Var_sh; G^1, ..., G^n|} compute the combined graph G_MG
compute the approximate local domain d for V in G_MG
foreach agent graph G^i ∈ MG do
    compute abstract graph A(G^i) w.r.t. d_i
return A(MG) = {|Var_sh; A(G^1), ..., A(G^n)|}
```

abstract model iff $\forall_{s'_1 \in [s_1]} \exists_{s'_2 \in [s_2]} s'_1 \longrightarrow s'_2$ in the concrete model. Correctness of the abstraction is proved by showing that the concrete model: simulates the *must* model, and is simulated by the *may* model.

4.3.1 Main Idea

In our case, concrete states are in the form of pairs $\langle l, \eta \rangle$. A natural choice is to define the equivalence relation w.r.t. variable evaluations and common locations. Arguably the simplest (non-trivial) example of such relation is achieved by clustering states that only differ in the evaluation of certain variables, which corresponds to the removal of a subset of variables $V \subseteq Var$ for the abstract model. In that way, we will cluster together the states $\langle l_1, \eta_1 \rangle$ and $\langle l_2, \eta_2 \rangle$ iff $l_1 = l_2$ and $\eta_1 =_{Var \setminus V} \eta_2$.

Additionally, we want the abstraction $\mathcal{A}$ to transform the MAS graph $MG = \{|Var_{sh}; G^1, \ldots, G^n|\}$ so that:

(i) its computation is *agent-based*, i.e., $\mathcal{A}(MG) = \{|Var_{sh}; \mathcal{A}(G^1), \ldots, \mathcal{A}(G^n)|\}$;
(ii) the abstract agent graphs $\mathcal{A}(G^i)$ preserve the same locations as in concrete G^i;
(iii) the changes must result from modifying variables V (e.g., their removal, domain restriction, etc.).

We will now systematically construct may-abstraction $\mathcal{A}^{may}(MG)$ (resp. must-abstraction $\mathcal{A}^{must}(MG)$) of MG, which will be given formally by Algorithms 1, 3 and 4.

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

The general structure of the procedure is shown in Algorithm 1. First, we approximate the set of reachable evaluations restricted to the subset of variables V in every location of the combined MAS graph G_{MG} by means of Algorithm 2, discussed in Section 4.3.2. 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 $V \cap Var^i$. This is implemented by function `ComputeAbstraction` (Algorithms 3 and 4), which will be presented in detail in Sections 4.3.3–4.3.5.

4.3.2 Approximating the Domains of Variables

Given a MAS graph MG, the approximation of reachable values for a set of variables $V \subseteq Var$ is defined in two variants.

Definition 4.6 (Upper-approximation)**.** *The **upper-approximation** of local domain is a function $d^+ : Loc \mapsto \mathcal{P}(Eval(Var))$ that maps each location $l \in Loc$ to the superset of its*

Algorithm 2: Approximation of local domain for $V \subseteq Var$

```
ApproxLocalDomain(G = G_MG, V)
1   foreach l ∈ Loc do
2       l.d := d_0
3       l.p := ∅
4       l.color := white
5   l_0.d := {η|_V | η ∈ Sat(g_0)}
6   Q := ∅
7   Enqueue(Q, l_0)
8   while Q ≠ ∅
9       l := ExtractMax(Q)
10      VisitLoc(l, V)
11      if l.color ≠ black then
12          foreach l' ∈ Succ^=1(l) do                 } enqueue immediate-neighbours
13              Q := Enqueue(Q, l')                    }
14              l'.p := l'.p ∪ {l}                     }
15          l.color = black
16  return {⟨l : (V ↦ l.d)⟩ | l ∈ Loc}
VisitLoc(l, V)
17  κ := l.d                                           } process incoming edges
18  foreach l' ∈ l.p, l' --g:α--> l do                 }
19      l.d := l.d ⊗ ProcEdge(l', g, α, l, V)          }
20  l.p = ∅                                            }
21  if κ ≠ l.d then                                    }
22      l.color := grey                                }
23  λ := l.d                                           } process self-loops
24  foreach l --g:α--> l do                            }
25      l.d := l.d ⊗ ProcEdge(l, g, α, l, V)           }
26  if λ ≠ l.d then                                    }
27      l.color := grey                                }
28      go to 23                                       }
ProcEdge(l, g, α, l', V)
29  δ_0 := {η ∈ Sat(g) | η|_V ∈ l.d}
30  let α := α^(1) ... α^(j)
31  for i = 1 to j do
32      δ_i := {η' = Effect(α^(i), η) | η ∈ δ_{i-1}}
33  return {η|_V | η ∈ δ_{max{j,0}}}
```

reachable evaluations, i.e. $\forall_{l \in Loc}\, d(l) \subseteq d^+(l)$ *as well as* $\forall_{l \in Loc}\, d(l)|_V \subseteq d^+(l)|_V$ *for any* $V \subseteq Var$.

Definition 4.7 (Lower-approximation). *The* ***lower-approximation*** *of local domain is a function* $d^- : Loc \mapsto \mathcal{P}(Eval(Var))$ *that maps each location* $l \in Loc$ *to the subset of its reachable evaluations, i.e.* $\forall_{l \in Loc}\, d^-(l) \subseteq d(l)$ *as well as* $\forall_{l \in Loc}\, d^-(l)|_V \subseteq d(l)|_V$ *for any* $V \subseteq Var$.

For an approximation of local domain $d^* : Loc \mapsto \mathcal{P}(Eval(Var))$, where $* \in \{+, -\}$, $Loc = Loc^1 \times \ldots \times Loc^n$, by d^*_i we denote a reduced to the i-th location component "narrowing" of d^*. Formally, for $1 \leq i \leq n$ and $l_j \in Loc^i$ the value of $d^*_i(l_j)$ is

defined as $\bigcup_{l \in Loc^1 \times \ldots \times Loc^{i-1} \times \{l_j\} \times Loc^{i+1} \times \ldots \times Loc^n} d^+(l)$ for the upper-approximation and $\bigcap_{l \in Loc^1 \times \ldots \times Loc^{i-1} \times \{l_j\} \times Loc^{i+1} \times \ldots \times Loc^n} d^-(l)$ for the lower-approximation.

Furthermore, when the set of variables $V \subset Var$ is clear from the context, we will sometimes use a shorter notation $d^*(l)$ (instead of $d^*(l)|_V$). Additionally, for $W \subseteq V$ (resp. $W \in V$) by $d^*(l, W)$ we denote the set of actual value tuples (resp. set of values), that is $\{\eta(W) \mid \eta \in d^*(l)\}$.

Detailed description of Algorithm 2. The function `ApproxLocalDomain` is parameterized by symbols d_0 and $\otimes$, such that $d_0 = \varnothing$ and $\otimes = \cup$ for the upper-approximation, and $d_0 = dom(V)$ and $\otimes = \cap$ for the lower-approximation.[10] On the input it takes the combined MAS graph G_{MG} and the set of variables $V \subseteq Var$, and then traverses its location using a modified version of the priority-BFS algorithm [Cor+09] in order to compute the chosen local domain approximation variant. The upper-approximation for every location $l \in Loc \setminus l_0$ is computed from $\varnothing$ by adding new, possibly reachable evaluations of V whenever they are produced on an edge coming to l. The lower-approximation for every location $l \in Loc \setminus l_0$ is computed from $dom(V)$ by iteratively removing evaluations of V that might be unreachable (i.e., only those that are commonly produced by all incoming to l edges are kept). For the initial location l_0 the computation starts from $\eta_0|_V$ in both cases and then proceeds normally.

In other words, starting from l_0 we systematically explore adjacent locations of the graph, and try to refine their current approximation with every (re-)visit until a stable approximation is obtained. Each location l must be visited at least once, and it shall be re-visited again whenever some of its predecessors l' gets their approximations $d^*(l')|_V$ updated. The computation halts when all locations were visited and have their approximation values stabilized.

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.[11] This shall reduce the number of potential re-visits in comparison with the generic FIFO variant of the queue.

Within the algorithm, with each location l we associate auxiliary attributes: $l.color \in \{white, grey, black\}$, the set of relevant predecessors $l.p \subseteq Loc \setminus \{l\}$, and the current approximation of the local domain $l.d$. The color indicates whether location has not been visited yet (*white*), had its approximation $l.d$ refined and awaits for neighbouring locations to be enqueued (*grey*), was 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 auxiliary attributes of every location are initialized with the *white* color, 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 of 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 separately with possible repetitions 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 α on evaluations satisfying the guard g and having their V counterpart in $l.d$. Lastly, if l changes its color to grey from either black or white, then all the immediate neighbours are enqueued to be inspected, adding l to the list of their relevant predecessors, and changing its color to black (lines 10–15).

[10] Note that in both cases, d_0 is simply a neutral element of the operation $\otimes$.

[11] The values of $r(l)$ can be derived from the reachability matrix produced by Warshall's algorithm, taking a row-wise sum of non-zero elements apart from those in the main diagonal.

$l \in Loc$	$r(l)$	$d^-(l, vt)$	$d(l, vt)$	$d^+(l, vt)$
$\langle$`idle,idle`$\rangle$	3	$\{0\}$	$\{0\}$	$\{0\}$
$\langle$`voted,idle`$\rangle$	2	$\varnothing$	$\{1,2,3\}$	$\{1,2,3\}$
$\langle$`obeyed,halt`$\rangle$	0	$\varnothing$	$\{1,2,3\}$	$\{1,2,3\}$
$\langle$`disobeyed,halt`$\rangle$	0	$\varnothing$	$\{1,2,3\}$	$\{1,2,3\}$

TABLE 4.1: Reachability index r of locations and reachable values of vt from: lower-approximation d^-, exact local domain d and upper-approximation d^+.

The algorithm halts and returns a stable approximation d (line 16) when the queue is empty and all the locations are *black*. Note that the subsequent approximations $l.d$ are weakly monotonic (i.e., $l.d \subseteq l.d'$ for d^+, and $l.d \supseteq l.d'$ for d^-). Since the sets of locations and edges are finite, and so are the variable domains, termination is guaranteed.

Example 4.5. *The local domain and its approximations obtained by* `ApproxLocalDomain` *for variable vt in the combined ASV graph of Example 4.2 can be found in Table 4.1.*

Theorem 4.1. *Given an extended MAS graph $G = G_{MG}$ and a set of variables $V \subseteq Var$, the approximations obtained by* `ApproxLocalDomain` *with $d_0 = \varnothing$, $\otimes = \cup$ corresponds to an upper-approximation, and with $d_0 = dom(V)$, $\otimes = \cap$ correspond to a lower-approximation of the local domain d with respect to V.*

Proof. Let d' and d'' be approximations obtained by `ApproxLocalDomain`(G_{MG}, V) with $d_0 = \varnothing$, $\otimes = \cup$ and $d_0 = dom(V)$, $\otimes = \cap$ respectively, and $\langle l, \eta \rangle$ be some reachable state of the model induced by G_{MG}. We will use an induction over the length $t \in \mathbb{N}^+$ of model's $M = \mathcal{M}(G_{MG})$ runs that end in a state induced by location l and show that:

(i) if $\exists \pi = \langle l_0, \eta_0 \rangle \dots \langle l, \eta \rangle \in Runs^t(M)$ then $\eta|_V \in d'(l)$;
(ii) if $\eta'|_V \in d''(l)$ then $\forall \pi = \langle l_0, \eta_0 \rangle \dots \langle l, \eta \rangle \in Runs^t(M). \eta' =_V \eta$.

Recall that by Definition 4.5, (exact) local domain d maps location $l \in Loc$ to the set of its reachable evaluations, or in other words to such evaluations $\eta \in Eval(Var)$ for which there exists an initial run in the induced model ending with $\langle l, \eta \rangle$.

Base case: $t = 1$ and $\pi = \langle l_0, \eta_0 \rangle$. From line 5 of `ApproxLocalDomain` it must be that $\eta_0|_V \in d'(l_0)$ as by construction of the algorithm any further changes of $d'(l_0)$ (in lines 19, 25) may only add new elements to it (but not remove the existing ones), and thus (i) holds. Analogously, (ii) holds because $d''(l_0)$ gets initialized to $\{\eta_0|_V\}$ (line 5) and further changes of $d''(l_0)$ (in lines 19, 25) may only remove its elements, i.e. final $d''(l_0)$ returned by `ApproxLocalDomain` is either equal to $\{\eta_0|_V\}$ or an empty set.

Inductive hypothesis: suppose that (i) and (ii) hold for all $t = 1 \dots k$, for some $k \geq 1$.

Inductive step: $t = k + 1$ and $\pi = \langle l_0, \eta_0 \rangle \langle l_1, \eta_1 \rangle \dots \langle l_t, \eta_t \rangle$ s.t. $l_t = l$ and $\eta_t = \eta$. By Definition 4.4, run π must be induced by some $l_0 \xrightarrow{g_1:\alpha_1} l_1 \xrightarrow{g_2:\alpha_2} \dots \xrightarrow{g_t:\alpha_t} l_t$, s.t. $\eta_{i+1} = Effect(\alpha_{i+1}, \eta_i)$ and $\eta_i \vDash g_{i+1}$ for $i = 0, \dots, t-1$. By IH (i) and (ii) must hold for $l_0, l_1, \dots, l_{t-1}$ and therefore $\eta_t|_V$ must appear on the output of `ProcEdge`$(l_{t-1}, g_t, \alpha_t, l_t, V)$ (possible in either case of upper- or lower-approximation), or `ProcEdge`$(l_{t-1}, g_t, \alpha_t, l_t, V) = \varnothing$ (possible in case of lower-approximation only). Hence, we can conclude that (i) and (ii) hold for $t = k + 1$ as well. □

Algorithm 3: Abstraction by variable removal

```
ComputeAbstraction(G = G^i, V, d = d_i)
1   ↪_a := ∅
2   foreach l -g:ch α-> l' do
3       foreach c ∈ d(l, V) do
4           g' := g[V = c]
5           α' = α[V = c]
6           ↪_a := ↪_a ∪ {l -g':ch α'-> l'}
7   ↪ := ↪_a
8   g_0 := g_0[V = η_0(V)]
9   Var^i := Var^i \ V
10  return G
```

4.3.3 Abstraction by Removal of Variables

The simplest form of abstraction consists in the complete removal of a given subset of variables $V \subseteq Var$ 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 $V' \subseteq V$ 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 $c' \in d(l, V')$, see Algorithm 3. The abstract agent graph obtained by removing variables V from G in the context of MG is denoted by $\mathcal{A}_{\{V\}}(G, MG)$. Whenever relevant, we will use $\mathcal{A}^{may}$ (resp. $\mathcal{A}^{must}$) to indicate the variant of the abstraction.

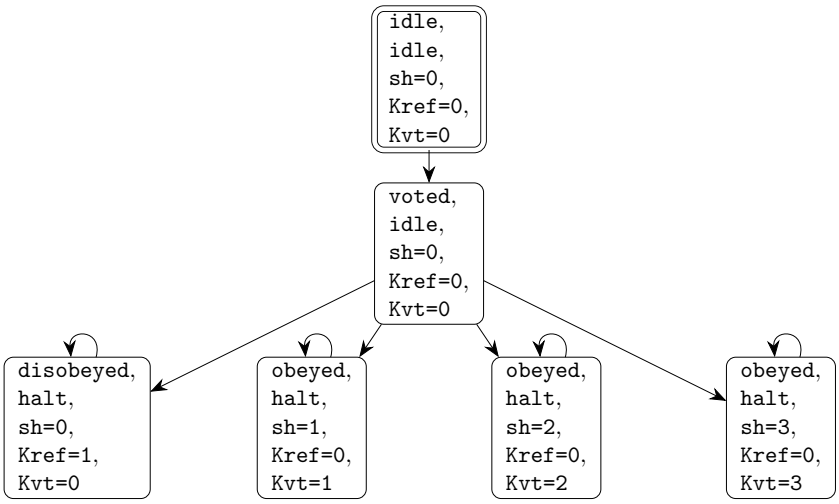


FIGURE 4.4: Unwrapping for $\mathcal{A}^{may}_{\{x\}}(ASV) = \{|Var_{sh};\ \mathcal{A}^{may}_{\{x\}}(G^{Voter}, ASV),\ G^{Coercer}|\}$.

Example 4.6. *The result of removing variable vt from the voter graph, according to the upper-approximation of the domain presented in Table 4.1, is shown in Figure 4.2b. Note that its unwrapping (shown in Figure 4.4) is distinctly smaller than the original one (Figure 4.3). Still, as we will formally prove in Section 4.4, all the paths of the model in Figure 4.3 are appropriately represented by the model in Figure 4.4.*

4.3.4 Merging Variables and Their Values

A more general variant of variable abstraction assumes a collection of mappings $F = \{f_1, \ldots, f_m\}$. Each mapping $f_i : Eval(X_i) \mapsto Eval(z_i)$ merges the local variables $X_i \subseteq Var^j$ of some agent graph G^j to a fresh variable z_i. The abstraction based on f_i removes variables X_i from graph G^j, and replaces them with z_i that "clusters" the values of X_i into appropriate abstraction classes. We will use $Args_R(f_i) = X_i$ and $Args_R(F) = \bigcup_{i=1}^{m} Args_R(f_i)$ to refer to the variables removed by f_i and F. $Args_N(f_i) = \{z_i\}$ and $Args_N(F) = \bigcup_{i=1}^{m} Args_N(f_i)$ refer to the new variables.

Note that the procedure in Section 4.3.3 can be seen as a special case, with a sole mapping f merging V to a fresh variable z with the singleton domain $dom(z) = \{\eta_0(z)\}$.

4.3.5 Restricting the Scope of Abstraction

The abstraction scheme can be further generalised by considering a set of mappings $F = \{(f_1, Sc_1), \ldots, (f_m, Sc_m)\}$, with each $f_i : Eval(X_i) \mapsto Eval(z_i)$ applied in some agent graph G^j, and $Sc_i \subseteq Loc^j$ defining the scope of f_i. That is, mapping f_i is applied only in the locations $l \in Sc_i$ by assigning $f_i(X_i)$ to z_i, and resetting the value of each $v \in X_i$ to v_0. Outside of Sc_i, the variables in X_i stay intact, and the new variable z_i 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 $\mathcal{A}_F(G, MG)$. Consequently, the abstraction of $MG = \{|Var_{sh}; G^1, \ldots, G^n|\}$ becomes

$$\mathcal{A}_F(MG) = \{|Var_{sh}; \mathcal{A}_F(G^1, MG) \ldots, \mathcal{A}_F(G^n, MG)|\}.$$

4.3.6 General Variant of the Abstraction

The general variant of $\mathcal{A}^*_F$, where $* \in \{may, must\}$, $F = \{(f_1, Sc_1), \ldots, (f_m, Sc_m)\}$ is described in Algorithm 4. In the main loop (lines 7–23) the edges of $G = G_j$ are transformed wrt the relevant pairs of (f_i, Sc_i) in the following way:

- edges entering or inner the Sc_i have their actions appended with (1) update of the target variable z_i and (2) update which sets the values of the source variables X_i to their defaults (resetting those),
- edges leaving or within the Sc_i have actions prepended with (1) update of source variables X_i (a temporarily one to be assumed for the original action) and (2) update which resets the values of the target variable z_i.

In order to make these transformations simultaneously, for each edge two auxiliary lists of mappings are derived: F_{trg}, which is relevant for edges entering or inner the scope, and F_{src}, which is relevant for edges leaving or within the scope.

Note that due to introduction of a scope, the variables from $Args_R(F)$ cannot be genuinely removed for a proper subset of locations — instead, their evaluation will be fixed to some constant value within the states, where location label is in the scope.

4.4 Correctness of Variable Abstraction

We will now prove that the abstraction scheme, proposed in Section 4.3, preserves the truth values of **ACTL*** formulae if the computation of variable domain d produces

Algorithm 4: General abstraction

`ComputeAbstraction(`G, d, F`)`

1 $X := Args_R(F)$
2 $Z := Args_N(F)$
3 $Fs := \{f_i \mid (f_i, Sc_i) \in F\}$
4 $Sc := \bigcup_{(f_i,Sc_i)\in F} Sc_i$
5 $g_0 := g_0 \wedge (Z{=}Fs(\eta_0(X)))$
6 $\hookrightarrow_a := \varnothing$
7 **foreach** $l \xhookrightarrow{g:ch\,\alpha} l'$ **do**
8 **if** $\{l, l'\} \cap Sc = \varnothing$ **then** } `out-of-scope edge`
9 $\hookrightarrow_a := \hookrightarrow_a \cup \{l \xhookrightarrow{g:ch\,\alpha} l'\}$ } `out-of-scope edge`
10 **else**
11 $F_{src} := \{f_i \mid (f_i, Sc_i) \in F \wedge l \in Sc_i\}$
12 $F_{trg} := \{f_i \mid (f_i, Sc_i) \in F \wedge l' \in Sc_i\}$
13 **foreach** $\eta \in d(l)$ **do**
14 $W_1 := \bigcup_{f\in F_{src}} Args_R(f)$ `// shall have the values assumed`
15 $W_2 := \bigcup_{f\in F_{src}} Args_N(f)$ `// shall have the values reset`
16 $Y_1 := \bigcup_{f\in F_{trg}} Args_R(f)$ `// shall have the values reset`
17 $Y_2 := \bigcup_{f\in F_{trg}} Args_N(f)$ `// shall have the values computed`
18 $g' := g[W_1 = \eta(W_1)]$
19 $\alpha' := (W_1 := \eta(W_1)).\alpha$
20 $\alpha' := (W_2 := (\eta_0(W_2))).\alpha'$
21 $\alpha' := \alpha'.(Y_2 := (F_{trg}(Y_1))(Y_2))$
22 $\alpha' := \alpha'.(Y_1 := \eta_0(Y_1))$
23 $\hookrightarrow_a := \hookrightarrow_a \cup \{l \xhookrightarrow{g':ch\,\alpha'} l'\}$
24 $\hookrightarrow := \hookrightarrow_a$
25 $Var := Var \cup Z$
26 **return** G

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***.

4.4.1 Simulations between Models

We first recall a notion of simulation [Mil71] between models, that preserves the truth values of **ACTL*** formulae [BK08; Cla+18; Coh+09].

Definition 4.8. *Let $M_i = (St_i, I_i, \longrightarrow_i, AP_i, L_i)$, $i = 1, 2$ 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** $\mathcal{R} \subseteq St_1 \times St_2$ 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 $s_1 \to s_1'$ then there is $s_2 \to s_2'$ 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 4.2. *If $(M_1, s_1) \preceq_{AP} (M_2, s_2)$, then for any* **ACTL*** *state formula ψ, using only propositions in AP, it holds that:*[12]

$$M_2, s_2 \models \psi \quad \textit{implies} \quad M_1, s_1 \models \psi \tag{$*$}$$

Proof. Suppose that $(M_1, s_1) \preceq_{AP} (M_2, s_2)$, and ψ is a state formula of **ACTL*** using only propositions in AP. We will show that ($*$) holds by induction over the structure of ψ.

Induction basis: Let $\psi = a$, where $a \in AP$. If $M_2, s_2 \models \psi$ holds, then $a \in L(s_2)$. Since $L_1(s_1) \cap AP = L_2(s_2) \cap AP$ by definition, it follows that $a \in L(s_1)$ and thus $M_1, s_1 \models a$.

Induction step: Suppose that ($*$) holds for state formulae ψ_1, ψ_2. Then:

(a) If $M_2, s_2 \models \psi_1 \wedge \psi_2$, then $M_2, s_2 \models \psi_1$ and $M_2, s_2 \models \psi_2$. Thus $M_1, s_1 \models \psi_1$ and $M_1, s_1 \models \psi_2$, therefore $M_1, s_1 \models \psi_1 \wedge \psi_2$.

(b) If $M_2, s_2 \models \psi_1 \vee \psi_2$, then $M_2, s_2 \models \psi_1$ or $M_2, s_2 \models \psi_2$. Thus $M_1, s_1 \models \psi_1$ or $M_1, s_1 \models \psi_2$, therefore $M_1, s_1 \models \psi_1 \vee \psi_2$.

(c) If $M_2, s_2 \models \mathsf{A}\varphi$, where φ is some path formula, then $\forall \pi \in \mathit{Paths}(s_2).\ M_2, \pi_2 \models \varphi$. Assume that $\exists \pi_1 \in \mathit{Paths}(s_1)\ .\ M_1, \pi_1 \not\models \varphi$ and therefore $s_1 \not\models \mathsf{A}\varphi$. Notice that using (i) and (ii-b) of Definition 4.8 we can inductively construct $\pi_2 \in \mathit{Paths}(s_2)$ corresponding to π_1, such that $\pi_1[i] \mathcal{R} \pi_2[i]$ for all $i = 1, \ldots, \mathit{len}(\pi_1)$. From (ii-b), all those pairs of states must satisfy the same set of atomic properties, it must be that $M_2, \pi_2 \not\models \varphi$ and thus $M_2, s_2 \not\models \mathsf{A}\varphi$, which is a contradiction.

□

Remark 4.4. *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.*

4.4.2 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 4.2. *Let $\alpha \in Act$, $V \subseteq Var$ and $\overline{V} = Var \setminus V$. Then for any $\eta_1, \eta_2 \in Eval(Var)$ such that $\eta_1 =_V \eta_2$ the following holds:*

$$\mathit{Effect}(\alpha, \eta_1) =_V \mathit{Effect}(\alpha[\overline{V}{=}\eta_1(\overline{V})], \eta_2)$$

Proof. Follows directly from Remark 4.1. □

Corollary 4.1. *Let $\alpha \in Act$, $V \subseteq Var$, $V = V' \cup V''$, $V' = V \cap lhs(\alpha)$, $V'' = V \setminus lhs(\alpha)$. Then for any pair of $\eta_1, \eta_2 \in Eval(Var)$ the following holds:*

$$\mathit{Effect}(\alpha[V = \eta_1(V)], \eta_2) \quad = \quad \big(\mathit{Effect}(\alpha, \eta_2[V' = \eta_1(V)])\big)\,[V'' = \eta_2(V'')]$$

[12] This is a known result from the literature, cf. [GL94; CGL94]. Nonetheless, to keep the dissertation self-contained, we present all relevant steps and include the proof, slightly adapted to our formalism.

Lemma 4.3. *Consider a pair of updates α and β, then for any $\eta \in Eval(Var)$ it holds that:*

$$Effect(\alpha \circ \beta, \eta) \quad =_{Var \setminus lhs(\alpha)} \quad Effect(\beta, \eta)$$

Furthermore, if $rhs(\alpha) \cap lhs(\beta) = \varnothing$, then the following holds as well:

$$Effect(\alpha \circ \beta, \eta) \quad =_{Var \setminus lhs(\beta)} \quad Effect(\alpha, \eta)$$

Additionally, if $Effect(\alpha, \eta) =_V Effect(\beta, \eta)$ for some $V \subseteq Var$, then for any γ it holds that:

$$Effect(\alpha \circ \gamma, \eta) =_V Effect(\beta \circ \gamma, \eta)$$

$$Effect(\gamma \circ \alpha, \eta) =_{V \setminus lhs(\gamma)} Effect(\gamma \circ \beta, \eta)$$

Proof. Follows directly from Remark 4.1 and Definition 4.3. □

Lemma 4.4. *Let $g \in Cond$, $V \subseteq Var$ and $V' = Var \setminus V$. Then for any $\eta_1, \eta_2 \in Eval(Var)$ such that $\eta_1 =_V \eta_2$ the following holds:*

$$\eta_1 \vDash g \quad \Leftrightarrow \quad \eta_2 \vDash g[V' = \eta_1(V')]$$

Proof. Follows from the fact that $\eta_1 \vDash g$ iff $g[Var = \eta_1(Var)]$ evaluates to $\top$. □

Corollary 4.2. *If $g \in Cond$, $Sat(g) = \{\eta_1, \ldots, \eta_k\}$, then $g \equiv \bigvee_{1 \le i \le k} \bigwedge_{v \in Var} (v = \eta_i(v))$.*

Definition 4.9. *Let $V \subseteq Var_1 \cap Var_2$, $s_i \in St_i$ and $s_i = \langle l_i, \eta_i \rangle$ for $i = 1, 2$. State s_2 **agrees with** a state s_1 over variables V (denoted $s_1 \simeq_V s_2$) iff $l_1 = l_2$ and $\eta_1 =_V \eta_2$.*

*Moreover, run $\pi_2 \in Runs(M_2)$ **agrees with** run $\pi_1 \in Runs(M_1)$ over variables V (denoted $\pi_1 \simeq_V \pi_2$) iff:*
(i) $len(\pi_1) = len(\pi_2)$, and
(ii) for every $1 \le i \le len(\pi_1)$, it holds that $\pi_1[i] \simeq_V \pi_2[i]$.

4.4.3 Variable Abstraction Is Sound

We prove now that the abstraction method from Algorithm 4, based on an upper-approximation of local domain d^+, indeed results with a simulation.

Let $MG = \{|Var_{sh}; G^1, \ldots, G^n|\}$, $G^i = (Var^i, Loc^i, l_0^i, g_0^i, Act^i, Effect^i, Chan^i, \hookrightarrow^i)$ and $G_{MG} = (Var_1, Loc, l_0, g_0, Act, Effect, Chan, \hookrightarrow)$, and (its abstraction) $\widehat{MG} = \mathcal{A}_F^{may}(MG)$, such that $G_{\widehat{MG}} = (Var_2, Loc, l_0, \widehat{g_0}, Act, Effect, Chan, \widehat{\hookrightarrow})$. The set $Var_1 = \bigcup_i Var^i$ comes from the combined MAS graph G_{MG}, the set Var_2 is equal to $Var_1 \setminus Args_R(F)$ in case of simple variable removal (which actually removes the variables from the model), and to $Var_1 \cup Args_N(F)$ in general case.[13] The set of atomic propositions AP_1 by definition can contain expressions over Loc and Var_1; for the subset $V_1 \subseteq Var_1$ the $AP_1(V_1) \subseteq AP_1$ is the subset of propositions without any occurrences of variables from $Var_1 \setminus V_1$; analogously, AP_2 contains propositions over Loc and Var_2, and for any $V_2 \subseteq Var_2$ the $AP_2(V_2) \subseteq AP_2$ is the subset of propositions without any occurrences of variables from $Var_2 \setminus V_2$.

Theorem 4.3. *Let $M_1 = \mathcal{M}(MG)$ and $M_2 = \mathcal{M}(\mathcal{A}_F^{may}(MG))$ be the models, such that $M_i = (St_i, I_i, \longrightarrow_i, AP_i, L_i)$ for $i = 1, 2$, $V = Args_R(F)$, $Z = Args_N(F)$. Then, for any $\widetilde{V} \subseteq (Var_1 \cap Var_2 \setminus V)$ the relation $\mathcal{R} \subseteq St_1 \times St_2$ defined by $\langle l_1, \eta_1 \rangle \mathcal{R} \langle l_2, \eta_2 \rangle$ iff $l_1 = l_2 \wedge \eta_1 =_{\widetilde{V}} \eta_2$ is a simulation relation over $AP = AP_1(\widetilde{V}) \cap AP_2(\widetilde{V})$ between M_1 and M_2.*

[13] Recall that in general case "removed" variables have their domain restricted to a singleton instead of being removed literally.

Proof ("simple" case). Here, we will first present a proof for a simpler case — variable removal; the proof for a general case is only technically more involved and shall be provided immediately thereafter.

Recall that g_0 is equivalent to $\bigwedge_{v \in Var_1}(v{=}\eta_0(v))$ and by Definition 4.1 $Sat(g_0) = \{\eta_0\}$ and $I_1 = \{\langle l_0, \eta_0 \rangle\}$. In case of the variable removal, we have $Var_2 = Var_1 \setminus V$ and $\widetilde{V} \subseteq Var_2$. By construction of the algorithm, the abstract $\widehat{g_0}$ is set to $g_0[V = \eta(V)]$.[14] Then, we can rewrite g_0 as $(\bigwedge_{v \in Var_2} v{=}\eta_0(v)) \wedge (\bigwedge_{w \in V} w{=}\eta_0(w))$ which is equivalent to $\widehat{g_0} \wedge (\bigwedge_{w \in V} w{=}\eta_0(w))$, meaning that $\forall_{\eta_1 \in Eval(Var_1)} \left(\eta_1 \vDash g_0 \Rightarrow \eta_1|_{Var_2} \vDash \widehat{g_0}\right)$. From this and the fact that the locations (both ordinary and initial) of MG and $\mathcal{A}_F^{may}(MG)$ are the same, we can conclude that $I_2 = \{\langle l_0, \eta_0|_{Var_2} \rangle\}$ and condition (i) of Definition 4.8 holds.

Now we show that condition (ii) of Definition 4.8 holds as well. By Theorem 4.1 we know that $d(l) \subseteq d^+(l)$, and therefore $d^+(l, V) \neq \varnothing$ (as otherwise, $d(l)$ would have to be an empty set as well). By construction of the algorithm, each (concrete) labelled edge $(l, labl, l') \in \hookrightarrow$ from MG must have a matching (abstract) labelled edge $(l, \widehat{labl}, l') \in \widehat{\hookrightarrow}$, where $\widehat{labl} = labl[V{=}c]$, for every $c \in d^+(l, V)$. Therefore, for any $\langle l, \eta_1 \rangle \mathcal{R} \langle l, \eta_2 \rangle$ and $\langle l, \eta_1 \rangle \longrightarrow_1 \langle l', \eta_1' \rangle$ that was induced by an edge $(l, labl_1, l') \in \hookrightarrow$, where $\eta_1 \vDash cond(labl_1)$ and $Effect(act(labl_1), \eta_1) = \eta_1'$, there must be $\langle l, \eta_2 \rangle \longrightarrow_2 \langle l', \eta_2' \rangle$ induced by the edge $(l, labl_2, l') \in \widehat{\hookrightarrow}$, where $labl_2 = labl_1[V{=}c]$ such that $c = \eta_1(V)$. By Lemma 4.4 $\eta_2 \vDash cond(labl_2)$ as $cond(labl_2) = cond(labl_1)[V{=}\eta_1(V)]$. By Lemma 4.2 $Effect(act(labl_2), \eta_2) =_{\widetilde{V}} Effect(act(labl_1), \eta_1)$ as $act(labl_2) = act(labl_1)[V{=}\eta_1(V)]$, which means that $\eta_2' =_{\widetilde{V}} \eta_1'$. Hence, we can conclude that $\langle l', \eta_1' \rangle \mathcal{R} \langle l', \eta_2' \rangle$ and condition (ii) from Definition 4.8 is fulfiled. □

Proof (general case). In general case, mappings from F can remove variables or merge them into new ones for the locations that are in scope. By construction of Algorithm 4, we have $Var_2 = Var_1 \cup Z$, where $Var_1 \cap Z = \varnothing$, $\widetilde{V} \subseteq Var_1 \setminus V$ and $\widehat{g_0} = g_0 \wedge g_Z$, where g_Z is equivalent to $\bigwedge_{z \in Z}(z{=}\eta_0(z))$.

For any pair of $\eta_1 \in Eval(Var_1)$ and $\eta_2 \in Eval(Var_2)$ such that $\eta_1 \vDash g_0$ and $\eta_2 \vDash g_Z$, by Lemma 4.4 it follows that $\eta_2[Var_1{=}\eta_1(Var_1)] \vDash \widehat{g_0}$ and therefore $\langle l_0, \eta_1 \rangle \in I_1$, $\langle l_0, \eta_2[Var_1{=}\eta_1(Var_1)] \rangle \in I_2$, meaning Definition 4.8(i) holds.

Now we will show that condition (ii) of Definition 4.8 holds as well. By construction of the algorithm, each (concrete) labelled edge $(l, (g, ch, \alpha), l') \in \hookrightarrow$ from MG must have a matching abstract edge $(l, (\widehat{g}_\eta, ch, \widehat{\alpha}_\eta), l') \in \widehat{\hookrightarrow}$ for every $\eta \in d^+(l)$, such that $\widehat{g}_\eta \equiv g[X'{=}\eta(X')]$ and for any $\widehat{\eta} \in Eval(Var_2)$ we have $Effect(\widehat{\alpha}_\eta, \widehat{\eta}) = (Effect(\alpha, (\widehat{\eta}[Z'{=}\eta_0(Z')])[X'{=}\eta(X')])[Z''{=}(F_K(X''))(Z'')])[X''{=}\eta_0(X'')]$, where:

- $J = \{i \mid l \in Sc_i\}$ denotes indices of relevant function mappings at the source-location, i.e., for which the source-location l is in the scope,
- $K = \{i \mid l' \in Sc_i\}$ denotes indices of relevant function mappings at the target-location, i.e., for which the target-location l' is in the scope,
- $X' = \bigcup_{i \in J} Args_R(f_i)$ denotes the "removed" variables at source-location,
- $X'' = \bigcup_{i \in K} Args_R(f_i)$ denotes the "removed" variables at target-location,
- $Z' = \bigcup_{i \in J} Args_N(f_i)$ denotes the "new" variables at source-location,
- $Z'' = \bigcup_{i \in K} Args_N(f_i)$ denotes the "new" variables at target-location.
- $(...[Z'{=}\eta_0(Z')])[X'{=}\eta(X')]$ substitution serves to reset values of the "new" variables at the source-location and "assume" the value of "removed" variables at the source-location,

[14] For the sake of brevity, we shall denote substitution of the form $[v{=}\eta_0(v) \mid v \in V]$ by $[V{=}\eta(V)]$.

- $(...[Z''{=}(F_K(X''))(Z'')])[X''{=}\eta_0(X'')]$ substitution serves to update the value of "new" variables and then reset the values of "removed" variables at the target-location.

Note that $Z', Z'' \subseteq Z$ and $X', X'' \subseteq V$, therefore for any $\widehat{\eta} \in \mathit{Eval}(\mathit{Var}_2)$ it holds that $(\widehat{\eta}[Z''{=}(F_K(X''))(Z'')])[X''{=}\eta_0(X'')] =_{\widetilde{V}} \widehat{\eta}$, and in particular $\mathit{Effect}(\widehat{\alpha}_\eta, \widehat{\eta}) =_{\widetilde{V}} \mathit{Effect}(\alpha, (\widehat{\eta}[Z'{=}\eta_0(Z')])[X'{=}\eta(X')])$. We know that $(\widehat{\eta}[Z'{=}\eta_0(Z')])[X'{=}\eta(X')] =_{\widetilde{V}} \widehat{\eta}$ and from $\mathit{Var}_1 \cap Z = \varnothing$ it follows that $\mathcal{V}(\alpha) \cap Z = \varnothing$. Then by Lemma 4.2 it must hold that $\mathit{Effect}(\alpha, (\widehat{\eta}[Z'{=}\eta_0(Z')])[X'{=}\eta(X')]) =_{\widetilde{V}} \mathit{Effect}(\alpha[X'{=}\eta(X')], \widehat{\eta})$.
Therefore, for any $\langle l, \eta_1\rangle \mathcal{R} \langle l, \eta_2\rangle$ and $\langle l, \eta_1\rangle \longrightarrow_1 \langle l', \eta_1'\rangle$ that was induced by an edge $(l, (g, ch, \alpha), l') \in \hookrightarrow$, where $\eta_1 \models g$ and $\mathit{Effect}(\alpha, \eta_1) = \eta_1'$, there must exist $\langle l, \eta_2\rangle \longrightarrow_2 \langle l', \eta_2'\rangle$ that was induced by the edge $(l, (\widehat{g}_\eta, ch, \widehat{\alpha}_\eta), l') \in \widehat{\hookrightarrow}$, where $\eta =_V \eta_1$. From $\langle l, \eta_1\rangle \mathcal{R} \langle l, \eta_2\rangle$ we know that $\eta_1 =_{\widetilde{V}} \eta_2$, and from Lemma 4.4 it follows that $\eta_2 \models \widehat{g}_\eta$. By taking $\widehat{\eta} = \eta_2$ we derive $\mathit{Effect}(\widehat{\alpha}_\eta, \eta_2) =_{\widetilde{V}} \mathit{Effect}(\alpha[X'{=}\eta_1(X')], \eta_2)$, and by Lemma 4.2 it follows that $\mathit{Effect}(\alpha[X'{=}\eta_1(X')], \eta_2) =_{\widetilde{V}} \mathit{Effect}(\alpha, \eta_1)$. Therefore, $\mathit{Effect}(\widehat{\alpha}_\eta, \eta_2) =_{\widetilde{V}} \mathit{Effect}(\alpha, \eta_1)$ and $\eta_2' =_{\widetilde{V}} \eta_1'$, which concludes that $\langle l', \eta_1'\rangle \mathcal{R} \langle l', \eta_2'\rangle$. □

We can now state our main theoretical result.

Theorem 4.4. *Let MG be a MAS graph, and F a set of mappings as defined in Section 4.3.5. Then, for every formula ψ of* **ACTL*** *that includes no variables being removed or added by F:*

$$\mathcal{M}(\mathcal{A}_F^{may}(MG)) \models \psi \quad \textit{implies} \quad \mathcal{M}(MG) \models \psi.$$

Proof. Follows directly from Theorems 4.2 and 4.3. □

4.4.4 Must-Abstractions of MAS Graphs

An analogous result can be obtained for must-abstraction $\mathcal{A}_F^{must}(MG)$.

Lemma 4.5. *Let MG be a MAS graph and d^- be the lower-approximation of a local domain defined for $V \subsetneq Var$. By the very nature of d^-, for any reachable location $l \in Loc$ it can have at most one element $|d^-(l)|_V| \leq 1$. Moreover, when $d^-(l)|_V = \{c\}$ there must exist reachable in $\mathcal{M}(MG)$ state $\langle l, \eta\rangle$, where $\eta(V) = c$.*

Theorem 4.5. *Let $M_1 = \mathcal{M}(MG)$ and $M_2 = \mathcal{M}(\mathcal{A}_F^{must}(MG))$, s.t. $M_i = (St_i, I_i, \longrightarrow_i, AP_i, L_i)$ for $i = 1, 2$, $V = \mathit{Args}_R(F)$, $Z = \mathit{Args}_N(F)$, $\widetilde{V} \subseteq \mathit{Var}_1 \cap \mathit{Var}_2$. Then, a relation $\mathcal{R} \subseteq St_2 \times St_1$, where $\langle l_2, \eta_2\rangle \mathcal{R} \langle l_1, \eta_1\rangle$ iff $l_2 = l_1 \wedge \eta_2 =_{\widetilde{V}} \eta_1$, is a simulation relation over $AP = AP_1(\widetilde{V}) \cap AP_2(\widetilde{V})$ between M_2 and M_1.*

The proof is analogous to that of Theorem 4.3; for the sake of completeness we report it below.

Proof. The construction of $\mathcal{A}_F^{must}(MG)$ implies that $\mathit{Var}_2 = \mathit{Var}_1 \cup Z$ and $\widehat{g_0}$ is equivalent to $(g_0[V{=}\eta_0(V)]) \wedge g_Z$, where g_Z is equivalent to $\bigwedge_{z \in Z} z{=}\eta_0(z)$. Therefore, for $\langle l, \eta_2\rangle \in I_2$ it holds that for any $\eta_1 \in \mathit{Eval}(\mathit{Var}_1)$ $\eta_1[\widetilde{V} = \eta_2(\widetilde{V})] \models g_0$ and $\langle l, \eta_1[\widetilde{V} = \eta_1(\widetilde{V})]\rangle \in I_1$, which means that Definition 4.8(i) holds.

As the abstraction is constructed in the same way, the reasoning about requirement Definition 4.8(ii) can be applied as stated in the proof of Theorem 4.3, except that by Lemma 4.5 each abstract edge $(l, (\widehat{g}_\eta, ch, \widehat{\alpha}_\eta), l') \in \widehat{\hookrightarrow}$, where $\eta \in d^-(l)$, is now matched by exactly one $(l, (g, ch, \alpha), l') \in \hookrightarrow$ from MG. □

Theorem 4.6. *Let MG be a MAS graph, and F a set of mappings as defined in Section 4.3.5. Then, for each formula $\psi \in$* **ACTL*** *including no variables removed or added by F:*

$$\mathcal{M}(\mathcal{A}_F^{must}(MG)) \not\models \psi \quad \textit{implies} \quad \mathcal{M}(MG) \not\models \psi.$$

Proof. Follows from Lemma 4.5 and Theorem 4.5. □

4.4.5 Abstraction on MAS Templates

When some agent graphs in the MAS graph are instantiations of a single template (i.e., they are identical up to variable renaming and evaluation of constant parameters), 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 4.10 (MAS template)**.** *A **MAS template** is a compact representation of a MAS graph MG as a tuple* $MT = (Var_{sh}, Const_{sh}; (GT^1, \#^1), \ldots, (GT^k, \#^k))$ *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* Var_{sh} *and shared constants* $Const_{sh}$*.*

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 $upsync(GT^i)$ 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 $lowsync(GT^i)$ that discards all the *edges with synchronisation labels* from GT^i.

Theorem 4.7. *Let MT be a MAS template, corresponding to the MAS graph MG. Then* $\mathcal{A}^{may}(upsync(MT))$ *induces a may-abstraction of MG, and* $\mathcal{A}^{must}(lowync(MT))$ *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. □

4.5 Complexity Analysis

Before further discussion and analysis of complexity results, we establish the auxiliary notation and list out general assumptions.

We assume random-access machine model of computation [Cor+09], where simple arithmetic operations ("+", "−", "∗", "/", "%"), data movement (load, store, copy) and control instructions all run in constant time. Furthermore, following a standard convention and assume that applying set-operations (intersection, union, difference) takes linear time in $O(N + M)$, where N and M are cardinalities of the operand sets.

Let $n := |Loc|$, $m := |\hookrightarrow|$, $k := |Var|$, $r := |V|$, where $V \subseteq Var$ is a subset of removed variables and $k \geq r$, $g_{var} := \max\{\mathcal{V}(cond(labl)) \mid \exists_{l,l' \in Loc}(l, labl, l') \in \hookrightarrow\}$ and $g_{len} := \max\{len(cond(labl)) \mid \exists_{l,l' \in Loc}(l, labl, l') \in \hookrightarrow\}$ for the maximal number of variables occurring in condition and maximal length of condition from given system labels, analogously $\alpha_{len} := \max\{len(upd(labl)) \mid \exists_{l,l' \in Loc}(l, labl, l')\}$ for maximal length of update associated with an action, and $d_{max} := \max\{|dom(v)| \mid v \in Var\}$ for the greatest variable domain cardinality.

A naive computation of set $Sat(g)$ for some $g \in Cond$ can be performed in $O(g_{len} \cdot d_{max}^{g_{var}})$ time. Without loss of generality, it can be assumed that the set of

initial valuations in $Sat(g_0)$ is precomputed and available in the form of look-up table, which guarantees that constant time lookup. Furthermore, note that would suffice to compute $Sat(g)$ for each $g \in Cond$ occurring within edge labels of the input graph G once. We shall assume that such it was pre-calculated and stored in advance, which can be done in $O(m \cdot g_{len} \cdot d_{max}^{g_{var}})$ time and $O(m \cdot d_{max}^{g_{var}})$ space. Additionally, for any atomic update $\alpha^{(i)}$ and $\eta \in Eval(Var)$ we shall assume that $Effect(\alpha^{(i)}, \eta)$ runs in $O(\alpha_{len})$.

4.5.1 Approximation of Local Domain (Algorithm 2)

The initialization loop on lines 1–4 runs in linear time $O(n)$, as involved three assignment statements take constant time. Thus, in line 5 it suffices to copy r-pointers, so it runs in $O(r)$ time. Using the generic heaps for priority queue with n elements, operations EXTRACTMAX and ENQUEUE are in $\Theta(\log n)$ and $\Theta(1)$ time respectively. Therefore, lines 1–7 take $O(n + r)$ to perform. Next, the loop in lines 8–15 requires each of n locations to be visited at least once. The condition for re-visit of location $l \in Loc$ from `VisitLoc`, which leads to location color $l.color$ changed to "grey", is the change of its temporal approximation value $l.d$. Note that `VisitLoc` algorithm maintains the invariant for every subsequent approximations for $l.d$ to be weakly monotonic w.r.t. subset inclusion for upper-approximation (resp. superset inclusion for lower-approximation). Hence, the approximation value of $l.d$ can change at most $\prod_{v \in V} |dom(v)|$ times, which is in $O(d_{max}^r)$. The line 9 runs in $O(\log n)$, lines 11-15 in $O(n^2)$ in the worst-case of densely connected graphs. Let `VisitLoc` run in T_1 time, then the whole block 8–15 can repeat $n \cdot d_{max}^r$ times resulting with $O(n \cdot d_{max}^r \cdot (T_1 + n^2))$ time, which yields that `ApproxLocalDomain` runs in $O(n + r + n \cdot d_{max}^r \cdot (\log n + T_1 + n^2))$.

The assignment statements in lines 17, 20, 22, 23, 27 are in $O(1)$. Note that in lines 21 and 26 it suffices to compare the cardinality of the sets, which can be done in $O(1)$.[15] The $\otimes$ is either a union or an intersection, which are both set operations running in $O(d_{max}^r)$. Upon visiting all n nodes, all m edges will get processed by `VisitLoc` in lines 18–19 and 24–25. The lines 26–28 run in $O(1)$ and will either reduce the number of potential re-visits of the location l (e.g., when the "`goto`" instruction gets executed multiple times) or increase the number of `ProcEdge` invocations (for self-loops) by a constant factor. Hence, visiting all locations d_{max}^r-times and having $n \cdot d_{max}^r$ calls to `VisitLoc` will invoke `ProcEdge` procedure $m \cdot d_{max}^r$ times. Let `ProcEdge` run in T_2 time, then by substitution of term $n \cdot d_{max}^r \cdot T_1$ with $m \cdot d_{max}^r \cdot (T_2 + d_{max}^r)$ we obtain that the `ApproxLocalDomain` runs in $O(n + r + m \cdot d_{max}^r \cdot (T_2 + d_{max}^r) + n \cdot d_{max}^r \cdot (\log n + 1 + n^2))$.

The lines 29–33 come down to computing potential outcome evaluations of V after taking α, namely the set $\{\eta'(V) \mid \eta' = Effect(\alpha, \eta) \wedge \eta \in Sat(g) \wedge \eta(V) \in l.d\}$, which takes $O(k \cdot \alpha_{len} \cdot d_{max}^r)$ time. This allows to conclude the running time of `ProcEdge`, $O(k \cdot \alpha_{len} \cdot d_{max}^r)$.

Finally, after the substitution of T_2 with $O(k \cdot \alpha_{len} \cdot d_{max}^r)$ we can conclude running time of `ApproxLocalDomain`.

Theorem 4.8. *The time complexity of the algorithm* `ApproxLocalDomain` *is* $O(k \cdot m \cdot \alpha_{len} \cdot d_{max}^{2r} + n^3 \cdot d_{max}^r)$, *or in* $O(m \cdot g_{len} \cdot d_{max}^{g_{var}} + k \cdot m \cdot \alpha_{len} \cdot d_{max}^{2r} + n^3 \cdot d_{max}^r)$ *in case if precomputation of look-up tables for* $Sat(g)$ *is taken into account, where* n *and* m *denote the total number of locations and edges within an input MAS graph,* k *the total number of its*

[15] The majority of the standard container implementations/objects provide a dedicated property that keeps track of the number of stored elements.

variables, r the total number of removed variables, g_{var} and α_{len} the maximal number of variables occurring in a condition label and the maximal length of update associated with an action respectively, and d_{max} the greatest variable domain cardinality.

In the `ApproxLocalDomain` we need $O(n)$ space for priority queue, $O(m \cdot d_{max}^r)$ for the look-up tables and $O(n \cdot (d_{max}^r + n + 1))$ for extending locations with additional attributes. The following theorem presents the resulting conclusion.

Theorem 4.9. *The space complexity of the algorithm* `ApproxLocalDomain` *is* $O((n^2 + m) \cdot d_{max}^r)$*, where n and m denote the total number of locations and edges within an input MAS graph, r the total number of removed variables, and d_{max} the greatest variable domain cardinality.*

4.5.2 Variable Removal Abstraction (Algorithm 3)

In the worst case the $V \subseteq Var^i$, when all intended for removal variables occur in G^i. The lines 1 and 7 run in constant time. The outer loop on lines 2–6 repeats m times, and the inner loop on lines 3–6 repeats at most d_{max}^r times. The substitutions on lines 4 and 5 run in $O(g_{len})$ and $O(k \cdot \alpha_{len})$ time respectively. The set union on line 6 runs in $O(m)$. The line 8 runs in $O(g_{len} \cdot r)$ and line 9 in $O(k)$. Therefore, we can conclude the following theorem.

Theorem 4.10. *The time complexity of the algorithm* `ComputeAbstraction` *is* $O(m \cdot d_{max}^r \cdot (g_{len} + k \cdot \alpha_{len}) + r \cdot g_{len} + k \cdot \alpha_{len} + k)$*, which is equivalent to* $O(m \cdot d_{max}^r \cdot (g_{len} + k \cdot \alpha_{len}))$*, where m denotes the total number of edges within an input MAS graph, k the total number of its variables, g_{len} and α_{len} the maximal length of a guard condition label and the maximal length of update associated with an action respectively, r the total number of removed variables, and d_{max} the greatest variable domain cardinality.*

Theorem 4.11. *The space complexity of the algorithm* `ComputeAbstraction` *itself is in* $O(m \cdot d_{max}^r)$*, where m denotes the total number of edges within an input MAS graph, r the total number of removed variables, and d_{max} the greatest variable domain cardinality.*

4.5.3 General Abstraction (Algorithm 4)

In case of general abstraction, we will first update the notation accordingly. Let $\hat{m} := |F|$ denote the number of function mappings, f_{len} maximal length of f_i, s.t. $(f_i, Sc_i) \in F$, and $r := |\bigcup_{i=1}^{\hat{m}} r_i|$ total number of removed/target variables, where $r_i = Args_R(f_i)$, and analogously $u := \sum_{i=1}^{\hat{m}} u_i$, where $u_i = Args_N(f_i)$ for new variables. Note that in general case $u \neq \hat{m}$, as new variable is not necessarily introduced.

The lines 1–3 run in $O(\hat{m})$, line 4 in $O(n^2)$, line 5 in $O(\hat{m} \cdot f_{len})$, line 6 in $O(1)$ time. The outer loop on lines 7–23 runs exactly m times, the "if-else" block condition involves set intersection and runs in $O(n)$. For the worst-case complexity analysis, we shall assume that the algorithm proceeds with "else" block (both line 9 and line 23 involve set union, but the latter will be repeated with each iteration of the for-loop on lines 13–23). The lines 11–12 take $O(\hat{m})$, and the inner loop on lines 13–23 repeats at most d_{max}^r times. An initialization of auxiliary sets on lines 14–17 runs in $O(\hat{m} \cdot (r + u))$ time. Line 18 takes $O(g_{len})$ time, lines 19 and 22 $O(r)$, lines 20–21 $O(f_{len} \cdot u)$ and line 23 $O(m^2)$ time. Line 24 is in $O(1)$ (e.g., by changing the pointer) and line 25 is in $O(k + u)$.

Therefore, we can infer the time complexity of the whole `ComputeAbstraction` as stated in the theorem below.

Theorem 4.12. *Algorithm* `ComputeAbstraction` *runs in* $O(\hat{m} + n^2 + \hat{m} \cdot f_{len} + m(\hat{m} + n + d_{max}^r(\hat{m} \cdot (r+u) + g_{len} + r + f_{len} \cdot u + m^2)) + k + u)$*, which is equivalent to* $O(md_{max}^r(\hat{m} \cdot (r + u + f_{len}) + g_{len} + m^2))$ *since* $n \in O(m)$*,* $u \in O(\hat{m})$*, where* m *denotes the total number of edges within an input MAS graph,* k *the total number of its variables,* $\hat{m}$ *the total number of function mappings,* r *and* u *the total number of removed and added variables respectively, and* d_{max} *the greatest variable domain cardinality.*

The F requires $O(\hat{m} \cdot (f_{len} + n))$ space. The sets W_1, Y_1 and W_2, Y_2 are in $O(r)$ and $O(u)$ space respectively. The new set of edges takes $O(m \cdot d_{max}^r)$ space. Our conclusion about the space complexity of `ComputeAbstraction` is expressed by the following theorem.

Theorem 4.13. *The space complexity of the algorithm* `ComputeAbstraction` *is* $O(m \cdot d_{max}^r + \hat{m} \cdot (f_{len} + n))$*, where* m *denotes the total number of edges within an input MAS graph,* k *the total number of its variables,* f_{len} *the maximal length of function mapping,* r *and* u *the total number of removed and added variables respectively, and* d_{max} *the greatest variable domain cardinality.*

4.5.4 Complexity in Practice

In many cases we can make the following assumptions:

- The number of removed variables is typically small, because removing many variables increases the chance of abstraction being too coarse for verification to be conclusive, and might even be infeasible due to computational complexity.
- Similarly, the number of possibly introduced variables and underlying expressions (from their mapping function) should be relatively small as well.
- The maximal number of variables occurring in guards as well as length should be relatively small, in general case having complex guards is considered a bad practice and degrades model readability; if it is absolutely necessary to include such compound guard conditions one should consider substitution by dedicated variable or a different specification formalism.
- The maximal length of updates should be relatively small. Analogous reasoning applies as in for the guards.

Formally, this amounts to assuming that the aforementioned parameters are bound by a constant a, namely:

$$\max\{r, u, f_{len}, g_{var}, g_{len}, \alpha_{len}\} \leq a \qquad (*)$$

Hence, we can conclude the complexity of algorithms `ApproxLocalDomain` and `ComputeAbstraction` in practice in terms of the following theorems.

Theorem 4.14. *Algorithm* `ApproxLocalDomain` *runs in* $O(a \cdot |Var| \cdot |\hookrightarrow| \cdot d_{max}^{2a} + |Loc|^3 \cdot d_{max}^a)$ *time and requires* $O((|Loc|^2 + |\hookrightarrow|) \cdot d_{max}^a)$ *space under assumption* $(*)$*, where* a *is some positive integer constant,* $|Var|$ *is the total number of variables of an input MAS graph,* $|Loc$ *is the total number of locations,* $|\hookrightarrow|$ *is the total number of edges, and* d_{max} *is the greatest variable domain cardinality.*

Theorem 4.15. *Algorithm* `ComputeAbstraction` *runs in* $O(d_{max}^a \cdot |\hookrightarrow| \cdot (|F| + |\hookrightarrow|^2))$ *time and requires* $O(|\hookrightarrow| \cdot d_{max}^a + |Loc| \cdot |F|)$ *space under assumption* $(*)$*, where* a *is some positive integer constant,* $|Var|$ *is the total number of variables of an input MAS graph,* $|Loc$ *is the total number of locations,* d_{max} *is the greatest variable domain cardinality, and* $|F|$ *is the total number of an input function mappings.*

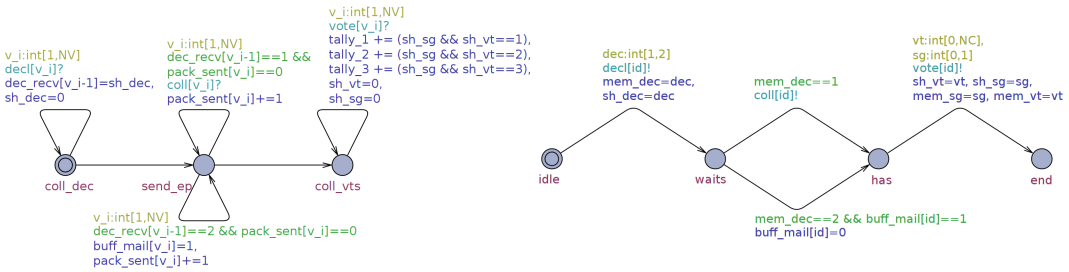


FIGURE 4.5: MAS graph for simplified postal voting: (a) Election Authority graph (left), (b) Voter graph (right).

Thus, under reasonable assumptions, the abstraction procedure runs in polynomial time and requires polynomial computing space.

Furthermore, from Lemma 4.1 we can conclude that by applying the variable removal abstraction the upper bound on the number of states of the yielded unfolding (of the abstract MAS graph) is reduced by an order of magnitude.

4.6 Case Study and Experimental Results

We evaluate our abstraction scheme on a simplified model of real-world scenario. 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. 4.5a) and *NV* instances of eligible Voters (Fig. 4.5b).

Each voter can vote for one of the *NC* candidates. The voter starts at the location `idle` 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 `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 `coll_dec`), distributes the packages (at `send_ep`), collects the votes, and computes the tally (at `coll_vts`). A vote is added to the tally only if the declaration is signed and the ballot is filled.

We will first present the results for *may*-abstraction — arguably, the more important case, since it can be used to prove an **ACTL*** formula true in a model — followed by the result for *must*-abstraction. 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. 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 `mem_dec`) at locations {`has`, `voted`}, and `dec_recv` at {`coll_vts`};
- Abstraction 3: combines Abstractions 1 and 2.

The abstract models were generated using a script in node.js.[16] The results were calculated by means of `verifyta` command line utility (v4.1.24), which is newer yet backwards compatible version that gives a more detailed summary. Time measurements were taken using the external tool (node.js), therefore for small time estimates there could be deviations due to the 'noises' caused by other factors.

[16] Implementation prototype and utilized models can be found at https://tinyurl.com/363pvpu5 and https://tinyurl.com/3eukkrkb.

conf	Concrete		Abstract 1			Abstract 2			Abstract 3		
NV,NC	#St	tv (sec)	ta (sec)	#St	tv (sec)	ta (sec)	#St	tv (sec)	ta (sec)	#St	tv (sec)
1,1	2.30e+1	0	0.03	1.90e+1	0	0.07	1.80e+1	0	0.16	1.60e+1	0
1,2	2.70e+1	0	0.03	2.10e+1	0	0.08	2.00e+1	0	0.06	1.70e+1	0
1,3	3.10e+1	0	0.03	2.30e+1	0	0.06	2.20e+1	0	0.05	1.80e+1	0
2,1	2.41e+2	0	0.02	1.41e+2	0	0.06	1.26e+2	0	0.06	9.30e+1	0
2,2	3.69e+2	0	0.02	1.77e+2	0	0.04	1.66e+2	0	0.03	1.06e+2	0
2,3	5.29e+2	0	0.02	2.17e+2	0	0.06	2.14e+2	0	0.04	1.20e+2	0
3,1	2.99e+3	0.01	0.02	1.14e+3	0	0.07	9.72e+2	0.01	0.05	5.67e+2	0
3,2	6.08e+3	0.01	0.02	1.62e+3	0.01	0.05	1.57e+3	0	0.04	6.93e+2	0
3,3	1.09e+4	0.04	0.02	2.20e+3	0.02	0.03	2.44e+3	0	0.05	8.38e+2	0.01
4,1	3.98e+4	0.12	0.02	9.57e+3	0.05	0.08	7.94e+3	0.03	0.08	3.54e+3	0.02
4,2	1.06e+5	0.55	0.01	1.52e+4	0.08	0.08	1.60e+4	0.05	0.06	4.62e+3	0.04
4,3	2.36e+5	0.95	0.01	2.26e+4	0.12	0.08	2.99e+4	0.07	0.08	5.94e+3	0.06
5,1	5.46e+5	1.48	0.02	8.17e+4	0.36	0.19	6.71e+4	0.18	0.25	2.23e+4	0.13
5,2	1.90e+6	6.42	0.02	1.43e+5	0.76	0.18	1.69e+5	0.50	0.23	3.09e+4	0.23
5,3	5.16e+6	24.95	0.02	2.30e+5	1.43	0.22	3.79e+5	1.16	0.22	4.21e+4	0.39
6,1	7.58e+6	31.34	0.01	7.03e+5	4.39	0.55	5.79e+5	1.92	0.44	1.41e+5	0.92
6,2	3.41e+7	170.25	0.01	1.34e+6	10.87	0.50	1.82e+6	7.64	0.40	2.07e+5	1.83
6,3	memout		0.01	2.31e+6	20.31	0.84	4.87e+6	22.67	0.40	2.97e+5	4.70
7,1	memout		0.01	6.05e+6	46.75	2.34	5.07e+6	22.16	1.91	8.89e+5	8.34
7,2	memout		0.02	1.25e+7	149.84	1.33	1.98e+7	107.95	2.01	1.38e+6	16.11
7,3	memout		0.02	2.28e+7	304.86	2.49	memout		2.35	2.08e+6	30.75
8,1	memout		0.02	5.20e+7	482.66	10.30	memout		8.04	5.61e+6	66.44
8,2	memout		0.19	memout		12.17	memout		7.58	9.15e+6	150.86
8,3	memout		0.07	memout		9.52	memout		7.99	1.44e+7	348.99
9,1	memout		0.12	memout		70.49	memout		64.96	3.53e+7	474.43
9,2	memout		0.06	memout		68.46	memout		71.69	memout	

TABLE 4.2: Experimental results for model checking of φ_{bstuff} in may-abstractions of postal voting.

4.6.1 Results for May-Abstraction

In the experiments for under-approximating abstraction, we verify the formula

$$\varphi_{bstuff} \equiv \mathsf{AG}(\sum_{\mathsf{i}=1}^{NC} \mathsf{tally}[\mathsf{i}] \leq \sum_{\mathsf{j}=1}^{NV} \mathsf{pack\_sent}[\mathsf{j}] \leq NV)$$

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 **ACTL*** formula φ_{bstuff} is satisfied in all considered instances of our voting model.

The results are presented in Table 4.2. 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 4.2).

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.

conf	Concrete		Abstract 1			Abstract 2			Abstract 3		
NV,NC	#St	tv (sec)	ta (sec)	#St	tv (sec)	ta (sec)	#St	tv (sec)	ta (sec)	#St	tv (sec)
1,1	23	0	0.03	15	0	0.07	14	0	0.08	14	0
1,2	27	0	0.03	15	0	0.05	14	0	0.06	14	0.01
1,3	31	0	0.03	15	0	0.05	14	0	0.04	14	0
2,1	241	0	0.01	81	0	0.04	70	0	0.04	70	0
2,2	369	0	0.01	81	0	0.02	70	0	0.03	70	0
2,3	529	0	0.03	81	0	0.02	70	0	0.04	70	0
3,1	2987	0	0.01	459	0	0.03	368	0	0.03	368	0
3,2	6075	0	0.02	459	0	0.03	368	0	0.03	368	0
3,3	1.09e+4	0	0.02	459	0	0.03	368	0	0.03	368	0
4,1	3.98e+4	0	0.01	2673	0	0.03	2002	0.01	0.04	2002	0
4,2	1.06e+5	0	0.01	2673	0	0.05	2002	0	0.03	2002	0
4,3	2.36e+5	0	0.01	2673	0	0.04	2002	0	0.03	2002	0
5,1	5.46e+5	0	0.01	1.58e+4	0	0.04	1.11e+4	0	0.06	1.11e+4	0
5,2	1.90e+6	0	0.01	1.58e+4	0	0.06	1.11e+4	0	0.05	1.11e+4	0
5,3	5.16e+6	0	0.02	1.58e+4	0	0.07	1.11e+4	0	0.05	1.11e+4	0.01
6,1	7.58e+6	0	0.01	9.40e+4	0	0.15	6.30e+4	0	0.09	6.30e+4	0
6,2	3.41e+7	0.01	0.01	9.40e+4	0	0.14	6.30e+4	0	0.10	6.30e+4	0
6,3	1.13e+8	0	0.01	9.40e+4	0	0.09	6.30e+4	0	0.09	6.30e+4	0
7,1	1.06e+8	0	0.01	5.62e+5	0.01	0.28	3.60e+5	0	0.24	3.60e+5	0
7,2	$\gg$1e+8	0	0.01	5.62e+5	0	0.34	3.60e+5	0	0.21	3.60e+5	0
7,3	$\gg$1e+8	0.01	0.01	5.62e+5	0	0.35	3.60e+5	0	0.23	3.60e+5	0
8,1	$\gg$1e+8	0	0.01	3.37e+6	0	0.90	2.08e+6	0	0.69	2.08e+6	0
8,2	$\gg$1e+8	0	0.02	3.37e+6	0	1.03	2.08e+6	0	0.63	2.08e+6	0
8,3	$\gg$1e+8	0	0.01	3.37e+6	0	0.86	2.08e+6	0	0.55	2.08e+6	0
9,1	$\gg$1e+8	0	0.01	2.02e+7	0	4.41	1.21e+7	0	2.43	1.21e+7	0
9,2	$\gg$1e+8	0	0.01	2.02e+7	0	2.80	1.21e+7	0	2.03	1.21e+7	0
9,3	$\gg$1e+8	0	0.01	2.02e+7	0	2.69	1.21e+7	0	1.99	1.21e+7	0
10,1	$\gg$1e+8	0	0.01	1.21e+8	0	9.61	7.03e+7	0	7.49	7.03e+7	0.01
10,2	$\gg$1e+8	0	0.01	1.21e+8	0	7.83	7.03e+7	0	8.02	7.03e+7	0
10,3	$\gg$1e+8	0	0.01	1.21e+8	0	8.99	7.03e+7	0	7.71	7.03e+7	0

TABLE 4.3: Experimental results for model checking of $\varphi_{dispatch}$ in must-abstractions of postal voting.

4.6.2 Results for Must-Abstraction

In the experiments for under-approximation, we used

$$\varphi_{dispatch} \equiv \mathsf{AG}\ \mathsf{coll\_vts}\ \mathit{imply}\ (\textstyle\sum_{\mathsf{j}=1}^{NV} \mathsf{pack\_sent}[\mathsf{j}] = NV)$$

expressing that the election packages must be eventually dispatched to all the voters.[17]

The results of the experiments are shown in Table 4.3. In all the completed cases, the verification of **ACTL*** formula $\varphi_{dispatch}$ on the abstract model was conclusive (i.e., the output was "false" for all the instances presented in Table 4.3).

In case when a AG formula is not satisfied by the model, its verification is in fact equivalent to finding a witness for a EF formula, which is often easy in practice. And it becomes even more so if model checker utilizes on-the-fly techniques (as is the case of UPPAAL) and examines the model simultaneously with the generation of states. This is confirmed by experimental results for model checking of $\varphi_{dispatch}$. However, it is worth noting that despite the lack of notable gains in terms of verification time from must-abstraction, the reduction in state space could be of immense importance when the model is generated prior to its exploration.

[17]The aforementioned formula is equivalent to $\mathsf{AF}(\sum_{\mathsf{j}=1}^{NV} \mathsf{pack\_sent}[\mathsf{j}] = NV)$ in our model; the former variant is used due to UPPAAL non-standard interpretation of the AF.

4.7 Related Work

State abstraction was introduced in the 1970s [CC77], and studied intensively in the context of temporal properties, see e.g. [CGL94; GJ02]. We discuss the most relevant works below.

[Cla+00a] presents an automatic iterative abstraction-refinement methodology CEGAR that is complete for the important fragment of **ACTL***. The method is described for the programs represented as BDDs, however according to the authors it is not tied to such and should be compatible with other representations. Intuitively, the procedure first partitions the program variables into the related variable clusters, which provides an initial equivalence relation for a coarse, memory-efficient starting abstraction. It then proceeds with model checking that abstract structure: either returning affirming that specification holds or obtaining an abstract counter-example run. In the latter scenario, it attempts to find a corresponding concrete counter-example run, and otherwise refines the abstraction and repeats the model checking step again. The procedure is guaranteed to halt. They also report an implementation in NuSMV and experimental results for model checking the industrial design of a multimedia processor from Fujitsu. Notably, the construction of the methods suggests that its implementation would have to be embedded into the model checking tool. Furthermore, by design the method takes the program on the input and using the current abstraction function (in terms of equivalence relations) generates the Kripke Structure. It is not discussed whether a final abstract structure can be expressed as the program or if its underlying abstraction function can be expressed in a human-readable format (probably not). In particular, the numerous iterations of refinement might be repeatedly computed each time when verifying multiple properties on the same program specification.

Other kinds of automatically generated lossless abstractions through abstraction-refinement include [DG18; SG04; Cla+03]. Unfortunately, lossless abstraction often results in abstract models that are still too large for practical verification.

[God14] proposes an abstraction method for 3-values temporal logics. It is similar to our work in that the abstraction is lossy and based on the may/must abstraction. The authors show that model checking 3-valued temporal logics has the same time and space complexities as in the case of corresponding conventional 2-valued logic. They present a procedure for an automatic abstraction using the generalized model checking, which does not restrict the supported temporal properties to the universal only, and, in theory, providing more precise verification results. Their work presents an improved approach to the existing methods of program verification, from which it inherits some fundamental assumptions. In particular, all the discussed representations — Kripke Structures (KS), Partial Kripke Structures (PKS), Modal Transition Systems (MTS) and Kripke Modal Transition Systems (KMTS), which generalizes the former three — are already of the low-level. In contrast, for our case we do not assume such representations to be immediately available: we consider a higher level input specification that is both more compact and human-readable. As consequence, our work takes into account and aims to address the potential issues associated with the top-down translation step of the model specification (e.g., by unfolding or unwrapping).

The [Coh+09] comes closest to our work. The authors present an abstraction technique for multi-agent systems, defined in terms of the interpreted systems framework, preserving temporal-epistemic properties (more precisely, the universal fragment

of **CTLK**). Given the equivalence relation, the local states and actions of the interpreted system are clustered, yielding the abstract system that simulates the original system. However, interpreted systems only consider the synchronous semantics and the local evolution function of an agent takes into account all the actions of other agents (i.e., the cardinality of the local evolution function is at least equal to the cardinality of the global evolution function). The authors do admit that choosing the collapsing equivalence relation (on local states and actions) is a challenge in itself. Note that validity of the abstract system relies on the equivalence relation properly chosen. Furthermore, the selected representation of the models by means of "vanilla" Interpreted Systems (IS) is debatable. Firstly, the set of local states of an agent can already be immense in itself.[18] This undermines the usability of the method, especially taking into account that the discussed abstraction procedure is not automated — automation and implementation were listed as the directions for future work. Secondly, in order to apply the proposed abstraction method *efficiently*, the sets of reachable local states have to be available. It is rarely the case that those sets are known in advance, whereas computation of reachable local states would often come down to the computation of the reachable global states. Basically, with IS-based representations the size of transition space is the same as in the global model (in terms of the asymptotic complexity), and the local state space is essentially a partition of the global one wrt the information available to a given agent.[19] Lastly, their abstraction technique focuses solely on simulations of the concrete model.

Other specific variants for multi-agent systems were also proposed in [ED08; LQR10]. Moreover, abstractions for strategic properties have been investigated in [AGJ04; BK06], and specifically for MAS in [KL17; BL17; BLM19]. However, in all those cases, the abstraction method is defined directly on the concrete model, i.e., it requires to first generate the concrete global states and transitions, which is exactly the bottleneck that we want to avoid.[20] Therefore, the main challenge we aim to address is completely omitted in those lines of work.

In this work, we focus on lossy may/must abstractions, based on user-defined equivalence relations that "cluster" concrete states into abstract ones. In contrast to abstraction studied in [DGG97; GHJ01; GJ02; God14], our method operates on modular (and compact) model specifications, both for the concrete and the abstract model. Data abstraction methods for infinite-state MAS [BLP11; BKL17] come somewhat close in that respect, but they still generate explicit abstract models. Moreover, they can be only used to *falsify* universal **CTL*** formulae, which is arguably the less interesting kind of approximation.

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 verification techniques.

4.8 Conclusions

In this chapter, we present a correct-by-design method for model reductions that facilitate formal verification of MAS. Theoretically speaking, our reductions are

[18] For example, the "card game" example [Coh+09] allows for 9 237 800 choices of initial state for each of the two players.

[19] In case of ISPL [LQR17], which extends the IS syntax with variable support, local states contain valuation of an agent's variables.

[20] [Coh+09; BLM19] use modular representations 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.

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 [AHK02; Sch04; Mog+14]. We also note that the procedure is generic enough to be used in combination with other techniques, such as partial-order reduction [Pel93; Ger+99; Jam+20a]. Finally, an implementation as an extension of the STV model checker [Kur+21] is considered.

Chapter 5

EASYABSTRACT: a Tool for Practical Model Reductions for Verification of Multi-Agent Systems

The experiments in Chapter 3 have shown that formal verification of voting procedures faces a substantial complexity barrier. This coincides with the fact that model checking of multi-agent systems (MAS) is known to be hard, both theoretically and in practice. The state-space explosion is a major challenge here, as faithful models of real-world systems are immensely huge and infeasible even to generate – let alone verify them. A smart abstraction of the state space may significantly reduce the model, and facilitate the verification. However, while state abstraction is well studied from the theoretical point of view, little work has been done on how to define actual abstractions in practice. 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 the 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 and a scenario of gossip learning for social AI.

5.1 Introduction

Multi-agent systems (MAS) [Woo02; SL09] describe interactions of autonomous agents, often assumed to be intelligent and/or rational. With the development of the Internet

and social networks, the impact of MAS on everyday life is becoming more and more significant. At the same time, their complexity is rapidly increasing. In consequence, formal methods for analysis and verification of MAS are badly needed.

Verification and model reduction. Algorithms and tools for verification have been in constant development for 40 years, with temporal model checking being most popular [BK08; Cla+18]. The main obstacle for *practical* use of those techniques is state-space explosion. Model checking of MAS with respect to their *modular representations* ranges from **PSPACE**-complete to undecidable [Sch03; Jam15]. A possible way to mitigate the complexity is by model reductions, such as abstraction refinement [Cla+00b] and partial-order reduction [Pel93]. Unfortunately, lossless reductions (i.e., ones that produce fully equivalent models) are usually too weak, in the sense that the resulting model is still too large for feasible verification.

Towards practical abstraction. In this work, we revisit the idea of lossy state abstraction [CC77; CGL94], and in particular *may/must abstraction* [GHJ01] that potentially removes relevant information about the system, but produces arbitrarily small reduced models. Such verification works best with users who are knowledgeable about the application domain, as its conclusiveness crucially depends on what aspects of the model are being removed. Ideally, the user should be a domain expert, which often implies no in-depth knowledge of verification algorithms. This calls for a technique that is easy to use and understand, preferably supported by a Graphical User Interface (GUI). Moreover, the abstraction should be *agent-based* in the sense that it operates on modular representations of the MAS, and does not require generating the full explicit-state model before the reduction. The theoretical backbone of our abstraction scheme was presented in Chapter 4 (and also discussed in [JK23a]). Here, we report on the implementation and show its usefulness through case studies.

Contribution. We propose a tool, EASYABSTRACT, for the reduction of MAS models by removing an arbitrary subset of variables from the model specification. After the user selects the variables to be removed, the tool can produce two new model specifications: one guaranteed to over-approximate, and one to under-approximate the original model. Then, the user can verify the properties of an original model by model checking the new specifications with a suitable model checker. Our model specifications are in the form of *MAS Graphs* (Definition 4.2), a variant of automata networks with asynchronous execution semantics and synchronization on joint action labels [Pri83; Jam+20a]. As the model checker of choice, we use UPPAAL [BDL04], one of the few temporal model checkers with GUI.

Our tool provides a simple command-line interface, where the user selects the input file with a model specification prepared in UPPAAL, the variables to be abstracted away, and the abstraction parameters. It outputs a file with the over- (resp. under-)approximating model specification, which can then be opened in UPPAAL for scrutiny and verification. The source code and examples are publicly available at `https://tinyurl.com/ez-abstract`. Importantly, the abstraction uses modular representations for input and output; in fact, it does *not* involve the generation of the global state space at all. To the best of our knowledge, this is the first tool for practical user-defined model reductions in model checking of MAS.

5.2 Formal Background

To specify the system to be verified, we use *MAS graphs* formalism [JK23a], based on standard models of concurrency [Pri83], and compatible with UPPAAL model specifications [BDL04]. The formal definitions of MAS graphs, their templates, unfolding

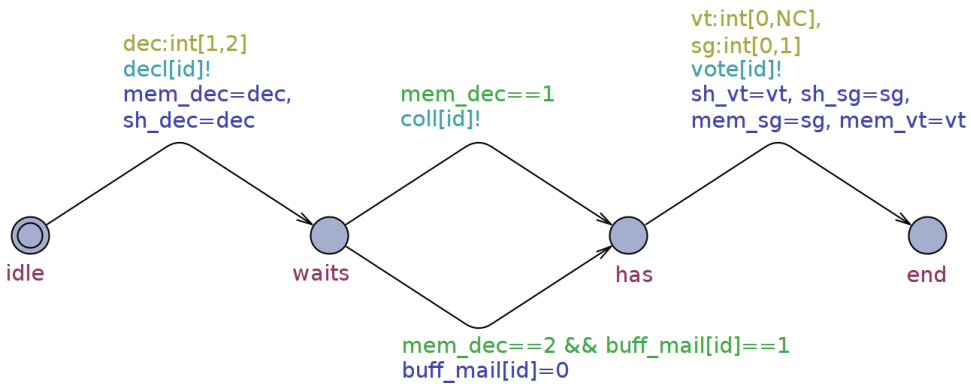


FIGURE 5.1: *Voter* template. The agent first declares if she prefers to receive the election package by post (`dec=2`) or in person (`dec=1`). Then, she waits until it can be collected, and casts the ballot together with her voting card. The *select* label for edge `idle`→`waits` (resp. `has`→`end`) specifies a nondeterministic choice of the value of variable `dec` $\in \{1,2\}$ (resp. `vt` $\in \{1,\ldots,NC\}$ and `sg` $\in \{0,1\}$).

and agent graphs can be found in Chapter 4. An example agent graph template, parameterized with a variable `id`, is shown in Fig. 5.1.

Recall that every MAS graph can be transformed to its *combined MAS graph*, and then unfolded into the *global model*, where states are defined by combined locations and valuations of all the variables. It is important to note that such models are usually huge and create an important bottleneck in model checking MAS.

Formal verification and model reduction. Our tool addresses model checking of temporal properties expressed in the well-known branching-time logic **CTL*** [Eme90], cf. Section 2.1 or [BK08] for a textbook on temporal model checking. To mitigate the impact of state space explosion, we use *state abstraction*, i.e., a method that reduces the state space by clustering similar *concrete states* into a single *abstract state*. In order for the scheme to be practical, it must be easy to use and avoid the generation of a concrete global model. We summarize the details of our abstraction scheme in the next section.

5.3 Abstraction by Removal of Variables

The simplest way to reduce a MAS graph is to remove some model variables or merge them into a new variable containing less information than the original ones. Our tool employs the abstraction scheme described in Chapter 4, and produces specifications of two abstract models: a *may-abstraction* (that over-approximates the concrete states and transitions) and a *must-abstraction* (that under-approximates them). Consequently, if a universal **CTL*** formula is true in the *may*-abstraction, then it must be true in the concrete model, and if it is false in the *must*-abstraction, then it must be false in the concrete model [JK23a].

For the sake of completeness, in the remainder of the section, we will now recall some notable forms of our variable abstraction from Chapter 4 in relation to the template shown in Fig. 5.1.

5.3.1 Variable removal

In the simplest variant, the abstraction concerns a complete removal of some variables $V \subseteq Var$ from the model specification. For example, one might remove variables

`mem_vt`, `mem_sg` from the agent graph in Fig. 5.1, i.e., the voter's memory of the cast vote and the voting declaration status. Selection of the right variables to remove requires a good understanding of the application domain; we assume that it is provided by the user. Roughly speaking, the abstraction procedure takes the combined MAS graph $comb(G)$, computes an approximation of the reachable values for every $v \in V$, and processes the edges of $comb(G)$ by substituting the occurrences of v at location ℓ with the values $u \in appr(v, \ell)$. If $appr(v, \ell)$ overapproximates (resp. underapproximates) the actual reachable values of v at ℓ, then the resulting model is a *may* (resp. *must*)-abstraction of G.

5.3.2 Variable merge and scoping

More generally, a subset of variables can be merged into a fresh variable by means of a user-defined mapping function. For example, `mem_sg` and `mem_vt` can be merged into a boolean variable `valid` given by `(mem_sg*mem_vt>0)`, indicating the validity of the vote.

Additionally, the user can specify the scope of abstraction, i.e., a subset of locations where the abstraction is applied, so that it only takes effect (and discards some details from the state) on a fragment of the system (when those details are no longer relevant/needed).

This should facilitate refining abstraction and help to obtain a conclusive result from the verification of an abstract model. For example, we could remove variable `mem_dec`, which encodes delivery medium, at locations `has` and `end`, after the election package was collected.

Depending on the model, property and user's creativity, one can come up with plenty of other interesting variants of abstraction with this tool. For example, when there is a set of indistinguishable agents of the same type (say Voters) and the property has to do with the capabilities of one or few agents only, then we could also use a coarser abstraction for all except one or few selected.

5.3.3 Abstraction on MAS templates

In some cases, approximation of variable domains on the combined MAS graph is computationally infeasible due to the size of the graph. An alternative is to compute it directly on the MAS template by the right approximation of the synchronization edges. On the downside, this sometimes results in largely suboptimal abstract models, i.e., ones more likely to produce inconclusive verification results.

The procedure may be run on either a combined MAS graph or just an agent template.[1] The former is expected to result in a more accurate approximation, but its computation also demands more resources (such as time and memory); the latter is less accurate but is also less resource-demanding. Unfortunately, the problem of state space explosion may occur even at the level of combined MAS graph representation. This becomes more evident on models with a large number of agents. The intuition/heuristic is to choose a template-based approximation, when the agent template does not have many synchronous edges or those synchronisations are not expected to have much impact on target variables evaluation. For example, the local domain for `mem_vt` and `mem_sg`, for which values are assigned locally (i.e., with no transitive reference to any global variable), could as well be approximated on the template instead; this

[1] A case of a partially combined MAS graph can also be defined, but that goes beyond the interest of the present discussion.

5.4 Architecture of EASYABSTRACT

The main components of the tool are: (1) local domain approximation and (2) generation of abstract model specifications. Furthermore, EASYABSTRACT allows performing simple pre-processing and code analysis. For convenience, execution options and flags can be saved in a configuration file. Each component can be called from the command line, possibly followed by a list of arguments:

- `configure`: sets the parameters in the configuration file;
- `unfold`: produces the combined MAS graph;
- `approx`: computes an approximation of the local domain;
- `abstract`: generates an abstract model specification based on the provided approximation of a local domain;
- `info`: lists the variables, locations, and edges in the model.

Unfold. Substitutes the constants with their values and converts the MAS graph from the input into the combined MAS graph.

Local domain approximation. Takes a subset of variables V, a target template ('ext' for the combined MAS graph) and an abstraction type $\mathrm{t} \in \{\text{upper}, \text{lower}\}$, and computes a t-approximation of the local domain over V. The result is saved to a JSON file, where location identifiers are mapped to an array of evaluation vectors. Note that the order of vector elements will correspond to an order of previously given variable identifiers.

Abstract model generation. Takes the mapping function with an upper-approximation (resp. lower-approximation) of the local domain, and computes the corresponding may-abstraction (resp. must-abstraction). The mapping function specifies the target agent name or template name, the scope of abstraction (a subset of location that should be affected), variables to be removed, and possibly a merge variable (name, initial value and evaluation expression). We assume that the input provided by the user is correct; some debugging might be added in the future.

Implementation details. The tool is written in `node.js`, which parses XML model specifications compatible with UPPAAL. The command line interface has been created with the help of the `yargs` library. After the model is parsed, its structure is processed using `antlr4`. Notably, using `antlr4` we can generate the dictionary of variable identifiers, their scope, domains, initial values, and associate expressions (e.g., occurring in edge labels) with their abstract syntax trees. The formal syntax is given by the EBNF grammar provided in file `yag.g4`. Additionally, a video demonstration is available at https://youtu.be/1tlyH1G9278.

5.5 Experimental Results

We have evaluated the tool by means of experiments on two benchmarks: a simple postal voting scenario and a gossip learning for social AI. The model specifications are available for download with the tool. The experiments were performed in combination with UPPAAL v4.1.24 (32-bit) on a machine with Intel i7-8665U 2.11 GHz CPU, running Ubuntu 22.04. We report the results for *may-abstractions*, typically more useful for universal branching-time properties.

#V	Concrete		Abstract (A1)		Abstract (A2)		Abstract (A3)	
	#St	**t**	**#St**	**t**	**#St**	**t**	**#St**	**t**
1	31	0	23	0	22	0	18	0
2	529	0.1	217	0.1	214	0.1	120	0.1
3	10 891	0.1	2203	0.1	2440	0.1	838	0.1
4	2.3e+5	0.9	22 625	1	29 938	0.1	5937	0.1
5	5.1e+6	25	2.3e+5	1	3.7e+5	1	42 100	0.6
6	memout		2.3e+6	20	4.9e+6	23	2.9e+5	5
7	memout		2.2e+7	304	memout		2.0e+6	33
8	memout		memout		memout		1.4e+7	357

TABLE 5.1: Verification of φ_{bstuff} on models with 3 candidates. $\#V$ is the number of Voter instances. We report the model checking performance for the concrete model, followed by *may*-models obtained by abstractions A1, A2, and A3.

5.5.1 Postal Voting

We use a scalable family of MAS graphs, which was originally proposed in [JRK22], to model a simplified postal voting system. The system consists of NV Voters, voting for NC candidates, and a single Election Authority, and proceeds in four subsequent phases: collection of voting declarations, preparation and distribution of election packages, ballot casting, and tallying. The verification concerns a variant of resistance to ballot stuffing, expressed by formula:

$$\varphi_{bstuff} \quad \equiv \quad \texttt{A[](b\_recv<=ep\_sent \&\& ep\_sent<=NV)}$$

where `b_recv` and `ep_sent` are variables storing the number of received ballots and sent election packages, respectively. For the experiments, we try the following abstractions:

A1: removes variables `mem_vt` and `mem_sg` from the Voter template, i.e., the voter's memory of the cast vote and the voting declaration status;

A2: removes variables `mem_dec` at Voter's locations {`has`, `voted`} and variable `dec_recv` at Authority's location {`coll_vts`}, i.e., the information about how the election package has been delivered;

A3: the combination of A1 and A2.

The results in Table 5.1 present the numbers of states in the global model generated during the verification, as well as the verification running times (in seconds), including the generation of abstract model specifications where applicable. Formula φ_{bstuff} is satisfied in all the reported instances; all three abstractions have been conclusive on it.

A more extensive study of the postal voting scenario will be presented in Chapter 6.

5.5.2 Social AI

The second series of experiments uses the specifications of gossip learning for social AI [Hea13; HDJ21], proposed in [KJS23]. The system consists of a ring network of AI agents, acting in three phases: data gathering, learning (based on the previously collected data), and sharing of knowledge. The goal of the agents is to collectively reach knowledge of quality $mqual \geq 2$. The system includes also an attacker who can

#Ag	Concrete		Abstract		
	#St	t	#St	Reduct	t
2	165	0	38	76.97	0
3	8917	0.1	555	93.78	0
4	4.6e+5	1.5	10 247	97.77	0.1
5	2.1e+7	123	1.5e+5	99.29	1.2
6	memout		2.8e+6	–	42
7	memout		4.1e+7	–	682
8	memout		memout		

TABLE 5.2: Verification of φ_{compr} on models of social AI. The column "#Ag" denotes the number of agents, and "Reduct" shows the level of reduction of the state space (in %).

impersonate any agent and fake its quality level. The model specifications are given as asynchronous MAS [Jam+20a; JPS20], and coded in the input language of the STV model checker [Kur+21], that shares many similarities with the input language of UPPAAL. After a straightforward manual translation to UPPAAL, we hardcoded the attacker's strategy to always share the lowest quality model, and verified formula:

$$\varphi_{compr} \equiv \texttt{A[](exists(i:int[1,NA])(impersonated!=i \&\& (!AI(i).wait || AI(i).mqual<2)))}$$

The formula says that, on all execution paths, at least one AI agent is compromised. The model checking performance is shown in Table 5.2. We have been able to conduct verification for concrete models with up to 5 agents (4 honest AI and 1 attacker), and up to 7 agents after applying a *may*-abstraction that discards all variables except for *mqual* in the AI template. It took less than 1s to perform a template-based over-approximation of a local domain followed by a generation of an abstract model.[2]

Chapter 8 will present a more extensive study of Social AI.

5.6 Related Work

The existing implementations of state abstraction for temporal model checking concern mostly automated abstraction. In particular, Counterexample-Guided Abstraction Refinement (CEGAR) [Cla+00b; Cla+03] has been implemented for NuSMV [Cim+02], and 3-valued abstraction [GHJ01; God14] was implemented in Yasm [GWC06] and YOGI [God+10]. In each case, abstraction involves the generation of the global state space, which is the main bottleneck when verifying MAS. Other, user-defined abstraction schemes have been defined only theoretically [SG04; BK06; DG18], and either require to generate all global states and/or transitions or assume an already alike low-level representation to be available. The approaches in [Coh+09; BLM19] come closest to our work, as they use modular representations of the state space. However, they both rely on a global representation of the transition space, and no implementation has been reported.

Conceptually, it was crucial for us that the abstraction method has the following features: user-friendly, supporting modular representation of the input and producing a compact and user-readable representation on the output, and not requiring generation of the global state space. Lack of existing methods that would satisfy these

[2] Used commands can be found in the demo video and on the github.

requirements motivated us to come up with our own solution. In the benchmarks, we report no comparison with other reduction techniques as, technically, for a sensible comparison it would have to be made with the methods that operate on the same (or least similar) pairs of input and output. As was already stated, to the best of our knowledge, there is no such, which leaves us with no candidates for the meaningful comparison.

5.7 Conclusions

We propose a tool for practical model reductions of multi-agent systems. The tool addresses state-space explosion by removal (either partial or complete) of selected variables from the model while preserving the truth of **ACTL** formulas. The experiments show significant gains in terms of verification time as well as memory, with minimal time used by the abstraction procedure. The procedure directly modifies the modular specification of the system, without generating the global model at all. Moreover, its output is open for further scrutiny and modifications by the user. While the must-abstraction was of little practical use, the may-abstraction demonstrated significant reduction of the state space and was sufficient for the verification of certain interesting properties.

In the future, we plan to extend our tool to abstractions preserving temporal-epistemic and strategic properties in combination with the MCMAS and STV model checkers [LQR17; Kur+21].

Chapter 6

Modelling and Verification of Polish Postal Voting of 2020

In the previous chapter, we mentioned brief experimental results, where variable abstraction was using to facilitate the verification of a postal voting scenario. Here, we focus entirely on this angle. To this end, we propose a formal model of a postal voting procedure, present formulas that capture important requirements, and show comprehensive experimental results from the verification.

Voting procedures are designed and implemented by people, for people, and with significant human involvement. Thus, one should take into account human factors in order to comprehensively analyse the properties of an election and detect threats. In particular, it is essential to assess how the actions and strategies of the involved agents (voters, municipal office employees, mail clerks) can influence the outcome of other agents' actions as well as the overall outcome of the election. In this chapter, we present our first attempt to capture those aspects in a formal multi-agent model of the Polish presidential election 2020. The election marked the first time when postal vote was universally available in Poland. Unfortunately, the voting scheme was prepared under time pressure and political pressure, and without the involvement of experts. This might have opened up possibilities for various kinds of ballot fraud, in-house coercion, etc. We propose a preliminary scalable model of the procedure in the form of a Multi-Agent Graph and formalize selected integrity and security properties by formulas of agent logics. Then, we transform the models and formulas so that they can be input to the state-of-art model checker UPPAAL. The first series of experiments demonstrates that verification scales rather badly due to the state-space explosion. However, we show that a recently developed technique of user-friendly model reduction by variable abstraction allows us to verify more complex scenarios.

6.1 Introduction

In the last 30 years, the world has become densely connected. Most IT systems address a complicated network of users, roles, functionalities, and infrastructure elements, often vastly distributed over geographical locations and cultural contexts. This results in a considerable space of potential threats, risks, and conflicting interests, that call for systematic (and preferably machine-assisted) analysis. What is more, IT services are implemented by people, with people, and for people. The intensive human involvement makes them hard to analyse beyond the usual computational complexity obstacles.

Voting procedures. Voting and elections are prime examples of services that are difficult to specify, hard to verify, and extremely important to society [HR16]. If democracy is to be effective, it is essential to assess and mitigate the threats of fraud, manipulation, and coercion [Men09; TJR16]. However, formal analysis of voting procedures must consider both the technological side of elections (i.e., protocols, architectures, and implementations) and the human and social context in which it is embedded [BRS16; Bas+17]. The impact of the social factor has become especially evident during the US presidential elections of 2016 and 2020. In 2016, individual voters were targeted before the election by a combination of technology and social engineering to induce emotional reactions that would change their decisions, and possibly swing the outcome of the vote (the Cambridge Analytica scandal). In 2020, a large group of voters was targeted after the election by unfounded claims that severely undermined the public trust in the outcome. In both cases, it is impossible to understand the nature of what happened, and devise mitigation strategies, without the focus on human incentives and capabilities.

Specification and verification of multi-agent systems. *Multi-agent systems (MAS)* provide models and methodologies for the analysis of systems that feature the interaction of multiple autonomous components, be it humans, robots, and/or software agents. The theoretical foundations of MAS are based on mathematical logic and game theory [SL09; Wei99; Woo02]. In particular, logic-based methods can be useful to formally specify and verify the outcomes of multi-agent interaction [DHM10; Eme90; Fag+95; Jam15].

Formal analysis with multi-agent logic is typically based on model checking [BK08; Cla+18]. The system is formalized through a network of graphs (or automata) that define its components, their available actions, and the information flow between them. The properties are usually given as *temporal properties*, expressing that a given temporal pattern must (or may) occur, or *strategic properties* capturing the *strategic abilities* of agents and their groups. Especially the latter kind of properties are relevant for MAS; e.g., one may try to capture voter-verifiability as the ability of the voter to verify her vote, and coercion-resistance as the inability of the coercer to influence the behaviour of the voter [TJR16]. There are many available model checking tools, though none of them is perfect. Some admit only temporal properties [BDL04; Dem+03; Kan+15], some focus on the less practical case of perfect information strategies [Alu+00; Che+13; LQR17], and the others have limited verification capabilities [Aki+20; KJK19; Kur+21]. Moreover, it is often unclear how to formalize an actual real-life scenario, including the "right" model of the system [Jam+20b] and the formal "transcription" of its desirable properties [Jam+21].

Socio-technical aspects of voting. In this work, we use agent-based methodology to propose and analyze a simple multi-agent model of an actual election, that combines the technological backbone of the voting infrastructure with a model of possible

human behaviors. The work is preliminary, in the sense that we do not explore the real breadth of participants' activities that might occur during the vote. Moreover, we mostly look at requirements that can be expressed as trace properties. This is because the computational complexity of the formal analysis turned out prohibitive even for such simple models and properties. We managed to mitigate the complexity through an innovative abstraction technique, but seeing if it scales well enough for realistic models of human interaction remains a subject for future work.

Case study: Polish postal vote of 2020. To focus on a concrete scenario, we consider the Polish presidential election of 2020. That was the first time when postal voting was universally available in Poland. Unfortunately, the voting scheme was prepared under pressure, and without the involvement of experts. This might have opened up possibilities for various kinds of ballot fraud, in-house coercion, etc. We propose a preliminary scalable model of the procedure in the form of a Multi-Agent Graph [BDL04; JK23a], and formalize selected integrity and security properties by formulas of agent logics. Then, we transform the models and formulas so that they can be input to the state-of-art model checker UPPAAL [BDL04], chosen because of its flexible model specification language and user-friendly GUI. As expected, the verification of unoptimized models scales rather badly due to the state-space explosion. To improve the performance, we employ a recently developed technique of user-friendly abstraction [JK23a], with more promising results.

Structure of the chapter. We begin by providing an outline of the Polish postal voting (PPV) procedure in Section 6.2. Then, the MAS Graph representation of the procedure, together with the formal translation of some important requirements of voting systems are presented in Section 6.3. This is followed by our experimental results and their discussion in Section 6.4. We discuss related work and related verification tools in Section 6.5, and give conclusions in Section 6.6.

6.2 Postal Voting Procedure

Postal voting is one of the oldest forms of voting. In its simplest version, it is easy to setup for the authorities and easy to follow for the voters. On the other hand, it can be susceptible to ballot fraud, lacks verifiability, and opens up potential for vote buying and coercion. What is more, only basic mechanisms of recovery are possible (e.g., cancelling the whole elections in case of irregularities). This sometimes leads to controversial ad hoc decisions when dealing with the irregularities [Hol21].

The postal voting procedure employed for the Polish Presidential Election in 2020 is no exception. There was an overall impression that the procedure had been prepared in haste [And20; Sku20] and with no proper research on the existing postal voting schemes that were proposed and used during the last two decades, such as [BRT13; KS19]. For example, voter authentication is based on the assumption that a voter's national identification number (PESEL) is secret, which is hardly the case in real life. Moreover, there are various ways of how authorities could delete votes, e.g., by sending invalid ballots to districts with anti-government majority [Sku20; Spo20; Fak20]. In this work, we make the first step towards systematic modelling and analysis of these kinds of threats.

The rules for organising the election of the President of Poland, with the possibility of postal vote, were published on June 2 [Pań20c] and June 3, 2020 [Mar20]; the date of the election was set to June 28, 2020. A complete list of legal acts defining the election procedure can be found in [Sej22; Pańd], see also [Pań20a; Pań20b; Min20b; Min20a] for additional details. For the postal vote, the regulations (mostly concerning

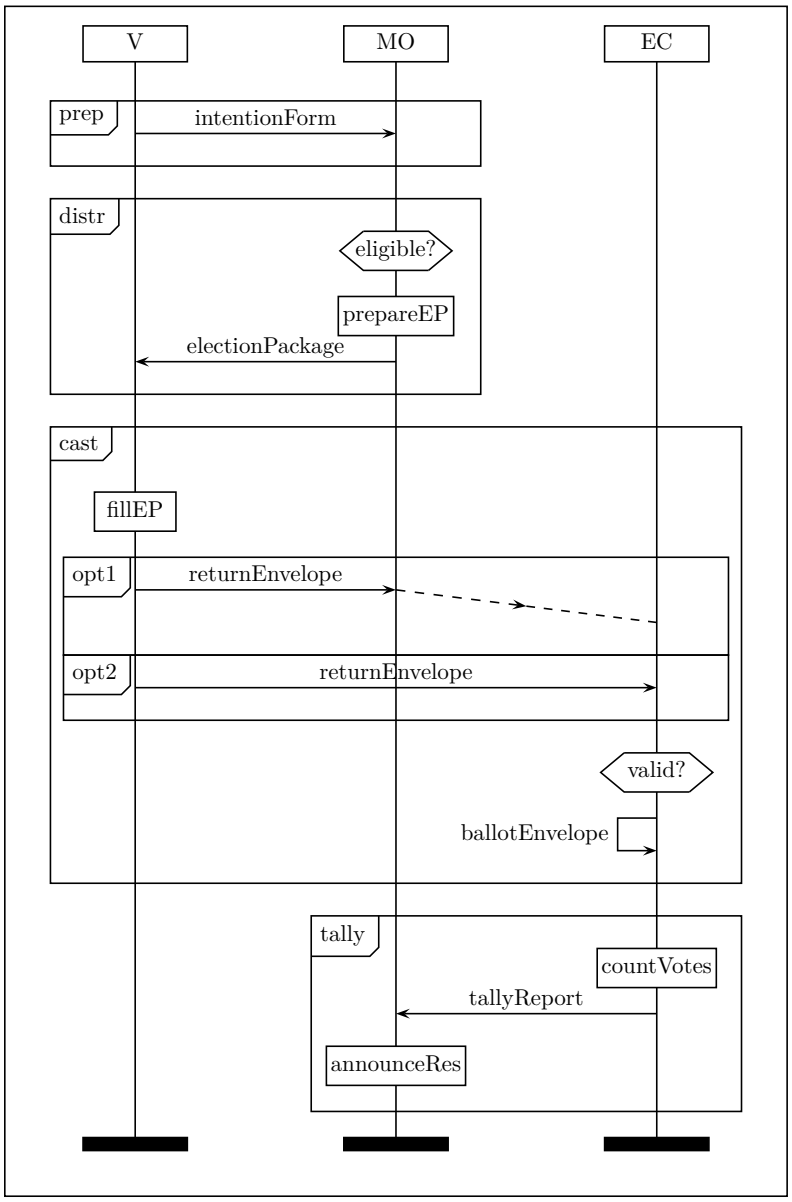


FIGURE 6.1: A simplified diagram of the voting process.

the time limits) vary based on the voter's location and her current quarantine status. We focus on the non-expat and non-quarantined voters,[1] but the protocol for the other types of voters is nearly the same.

The protocol consists of several, partly overlapping phases: the setup which involves expression of intention to vote by post, preparation and distribution of election packages (EPs), casting of the vote, validation of votes on the election day, and tallying, see Figure 6.1.

Setup. A voter expresses her intention to vote by post to its local municipal office (MO) at the latest 12 days before the day of an election (EDay). This can be done in either oral, written, or electronic form. The intent expression must contain the voter's personal information, such as full name, DOB, ID number (PESEL), phone number,

[1] Changing this to another type of voter (or even adding extra ones) would only require changing the values of a few configuration variables. However, this would also add unnecessary (in the context of the satisfiability of requirements being discussed) complexity to the model.

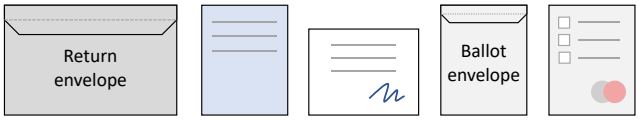


FIGURE 6.2: EP content: return envelope, instruction card, voting card, ballot envelope, stamped ballot.

email, and residential and postal addresses. If the voter prefers to collect the EP in person, this must be specified instead of a postal address. Additionally, the voter can request to change the municipality assigned to her in the voters' register once before the start of the election.[2] It is also possible to obtain a voter certificate, which when provided allows one to vote in any election commission of any municipality, however obtaining such will lead to a voter being crossed out from the voters' lists, allowing for in-person voting only.

EP preparation. Upon receipt of the intention, a municipal office employee checks the voters' register and prepares and distributes the EP, provided that the applicant is an eligible voter and no required information is missing. A complete EP (Figure 6.2) must contain an instruction, a ballot stamped by both the National Electoral Commission (PKW) and the local electoral commission, a voting card, and two envelopes: one for the ballot and one to be returned. The EPs must be delivered or made available for collection to voters no later than five days before the EDay.

Casting. When a complete EP is collected, the voter should put a single 'X' mark against the preferred candidate, put the ballot into a ballot envelope, sign a voting card and place it together with a ballot envelope into a return envelope. Both ballot and return envelope must be sealed. If there is a deviation in any of the above steps (e.g., the ballot envelope is not sealed), this would invalidate the casting of the vote. Then, the voter must either send the filled REnv to MO, where it will be stored until it is passed to the electoral commission (EC) on the EDay, or turn it in to the assigned electoral commission.

Validation. The EC has to print the voters' list (partitioned according to their municipality) one day before the election at the latest. This will be used to check the validity of the vote and make sure that no person can vote multiple times. In particular, people who requested and obtained a voter's certificate – a document which allows a voter to go to any election commission for in-person voting (and not necessarily the one assigned to him) – are not allowed to vote by post. If the voter is eligible and the REnv is complete, the BEnv is put into the ballot box.

Tallying. At the end of the EDay, when all REnvs are collected, the commission opens the BEnvs and prepares a voting protocol with information on the number of received REnvs, invalid votes, and the local tally. The protocol is sent to MO which checks it using proprietary software. If the errors are within the margins allowed by legislation, the MO accepts the protocol. Otherwise, the protocol is rejected, and the electoral commission must prepare a new one. When all protocols are accepted and merged, the final tally and the winner are publicly announced.

[2]Voters' register contains a list of eligible voters, their full name, date of birth, ID number and residence address. The register is maintained by the municipal office, which can also make changes on request. To facilitate access by electoral commissions during the election, it is partitioned into voter lists based on the residence address. Its main purpose is to enable verification of voters' eligibility and to ensure that each voter can vote at most once.

(a)

(b)

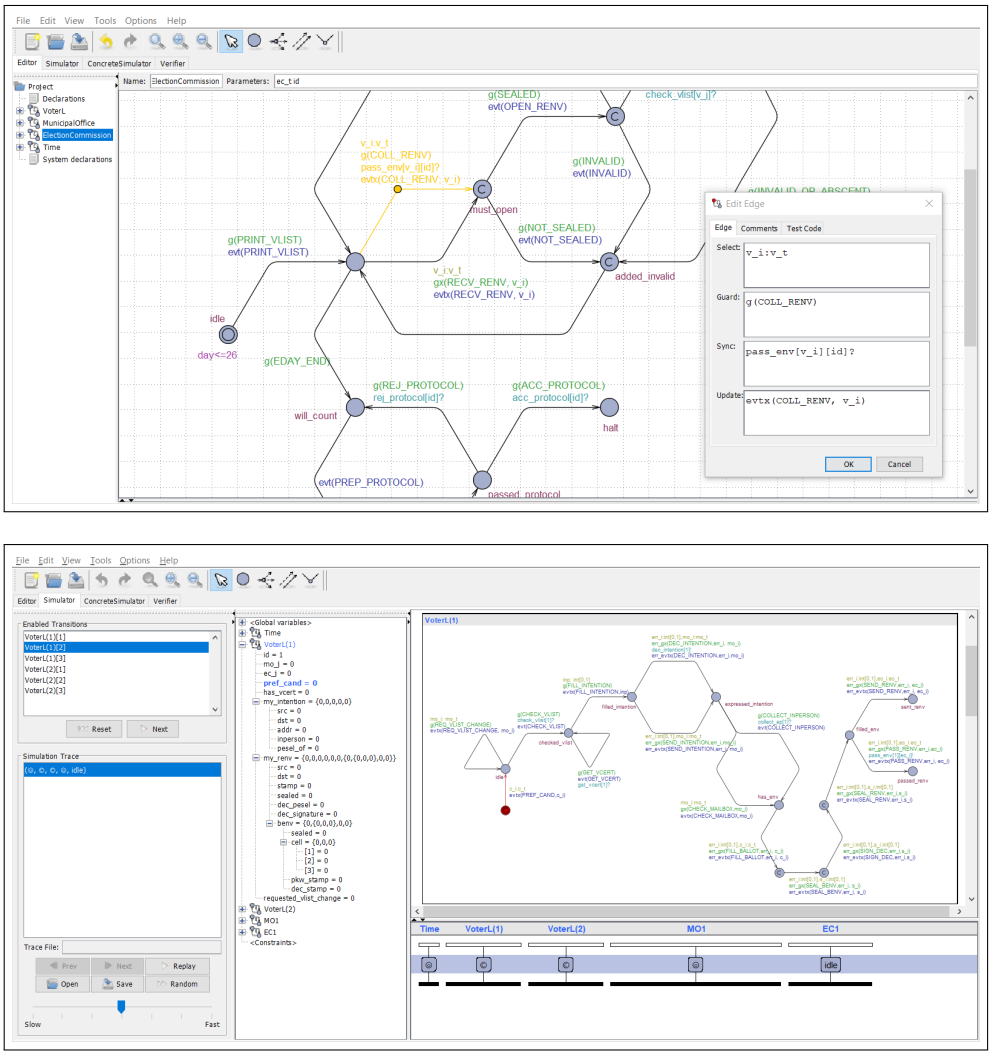


FIGURE 6.3: UPPAAL GUI: (a) editor tab, (b) simulation tab.

6.3 Formal Model of the Procedure

We can now present our preliminary model of the postal voting protocol. The code of the model is available at https://github.com/polishpostalvote2020/model.

We chose the UPPAAL model checker as the modelling environment because of its user-friendly GUI (see Figure 6.3) and flexible system specification language. As previously discussed in Chapter 3, the GUI is especially important because it allows a preliminary validation of a system specification even at the early stages of modelling.[3] A system in UPPAAL is represented as a (parameterized) network of (parameterized) finite automata [BDL04]. The parametrization can be used to define a set of almost identical processes easily. In order to represent an occurrence of events, as well as define the available strategies of participants, we use the extension of automata networks to *MAS graphs*, proposed in Chapter 4.

The MAS graph for the PPV procedure consists of the following agent templates:

- Voter (V), depicted in Figure 6.4;
- Electoral Commission (EC), depicted in Figure 6.6;
- Municipal Office (MO), depicted in Figure 6.5;
- Time counter, depicted in Figure 6.5.

The numbers of agent instances are denoted by *NV*, *NMO*, and *NEC*. Their values, together with the number of candidates *NC* are specified as part of the model configuration. The sole Time agent is used to impose discrete time constraints on the actions of other agents.

We extend the base model (having infallible agents only) by specifying the following mistakes for Voter - may initiate communication and send the forms to a wrong MO or EC, may leave ballot or return envelopes unsealed, misplace the cross mark on the ballot, forget to fill or sign the voting card, attempt to vote by post after obtaining voter's certificate, for Municipal Office (only in some explicitly specified experiments) - may prepare and then distribute ballots without a proper stamp, invalidating those. This yields a hierarchy of the (partially ordered) models, allowing us to study the satisfiability of properties on a finer-grained level, and furthermore get a better understanding of the potential impact of human errors on the system.

The agent templates include a variety of behaviours that can result from human errors as well as purposeful misbehaviour. For example, a Voter may not fill the forms properly, or attempt to communicate with the wrong municipal office or electoral commission. Furthermore, a Municipal Office can send out an invalid ballot by using a photocopied rather than a genuine stamp (which actually happened during the election). In this version, we do not explicitly model the postal services, and thus omit the possible malicious or erroneous actions of postal clerks, or an adversary.[4] Instead, we focus on an analysis of human interactions and the possible effects of their mistakes (as deviation from the expected behaviour). Similarly, we omit rare events, e.g., those that involve the power of attorney for in-person hand-in of the REnv.

[3] In UPPAAL, system components are defined together with their graph-like (local) representation, which could greatly facilitate in reducing the number of bugs/errors, improve understanding and presentation of the model, and provide more confidence that *we know what we are modelling and if it is actually what wanted to model*.

[4] It is worth noting that having a modular representation allows extending the model with other types of agents, including adversary, if needed.

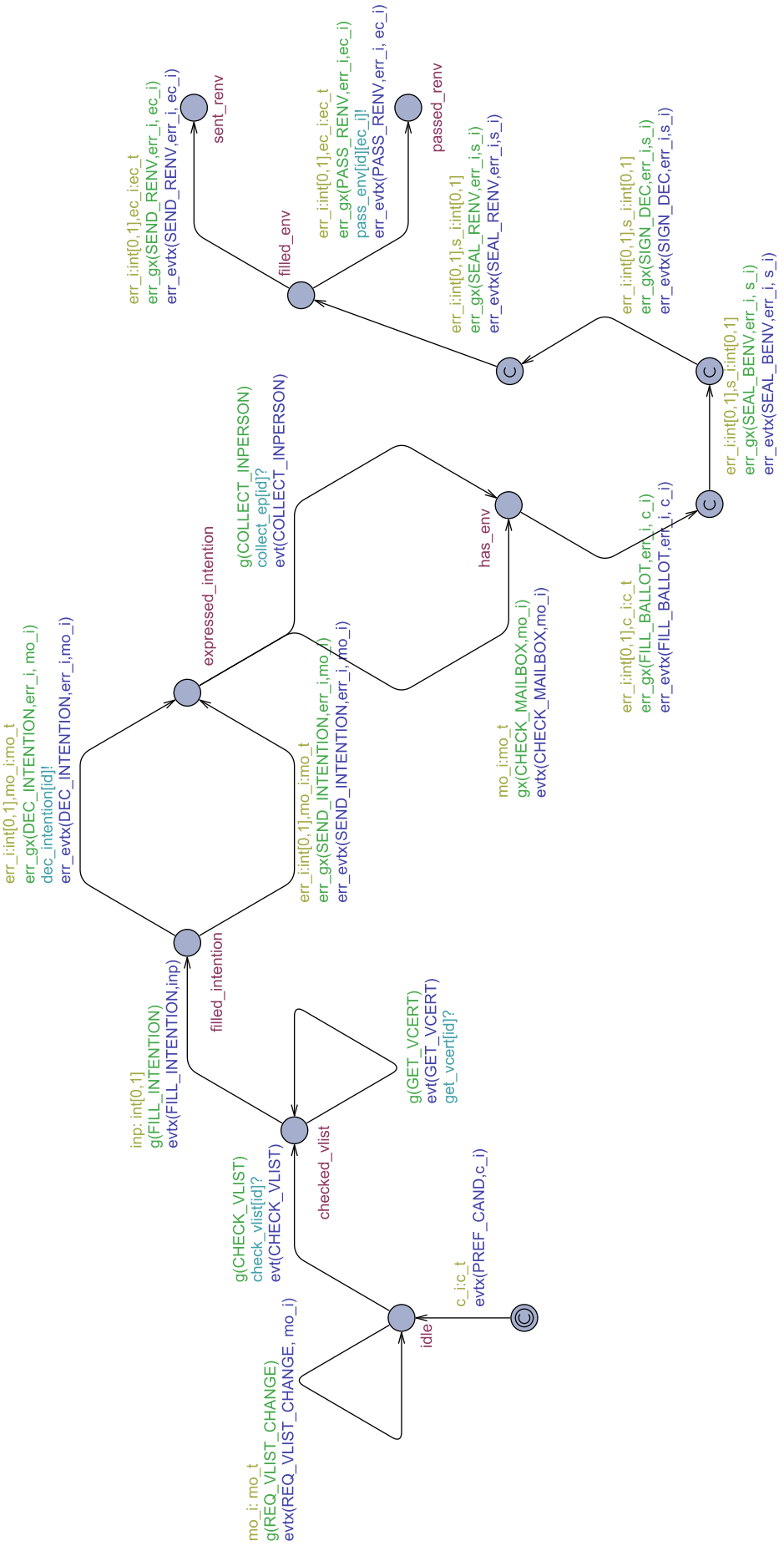


FIGURE 6.4: Voter template.

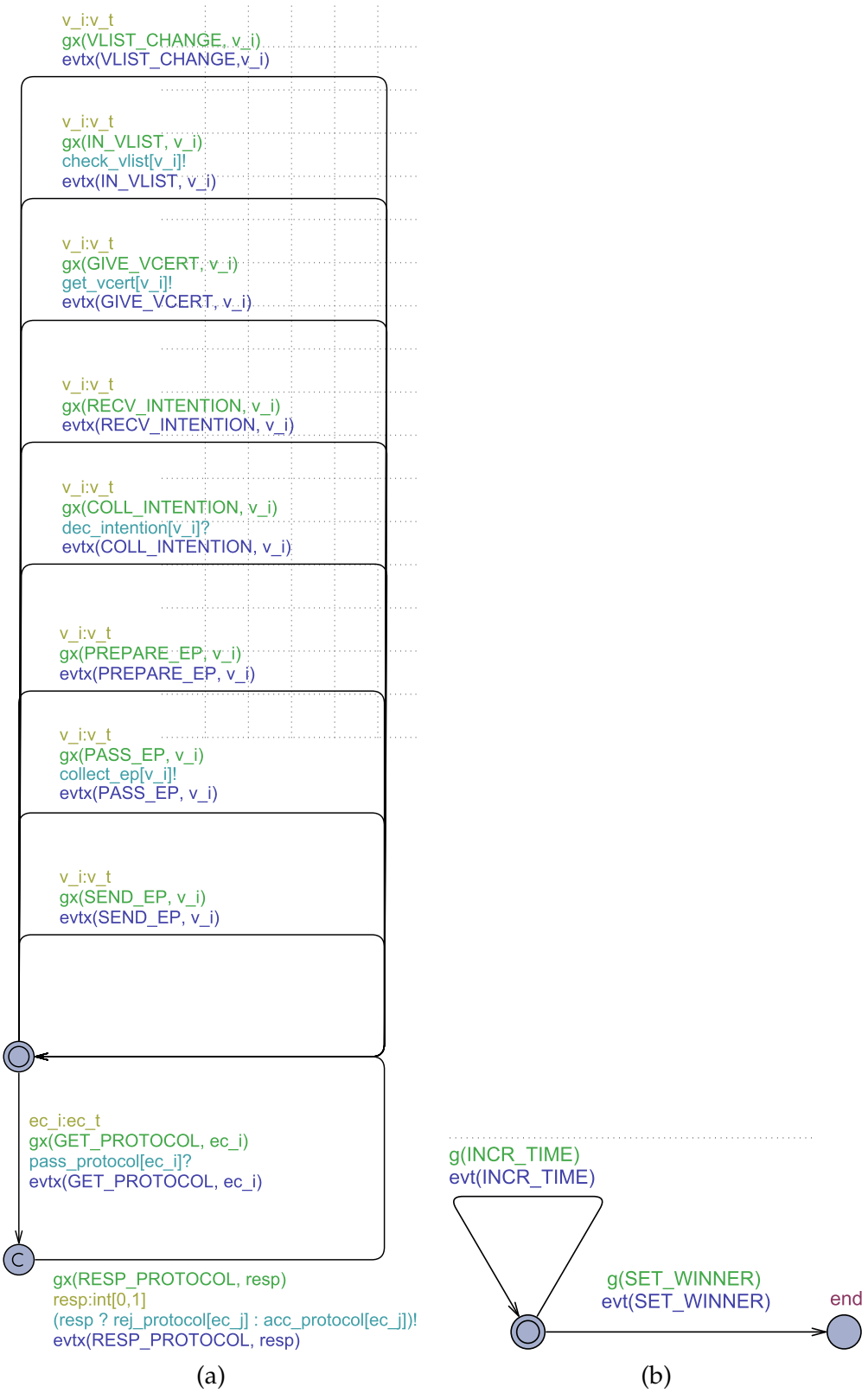


FIGURE 6.5: (a) Municipal Office template, (b) Time singleton.

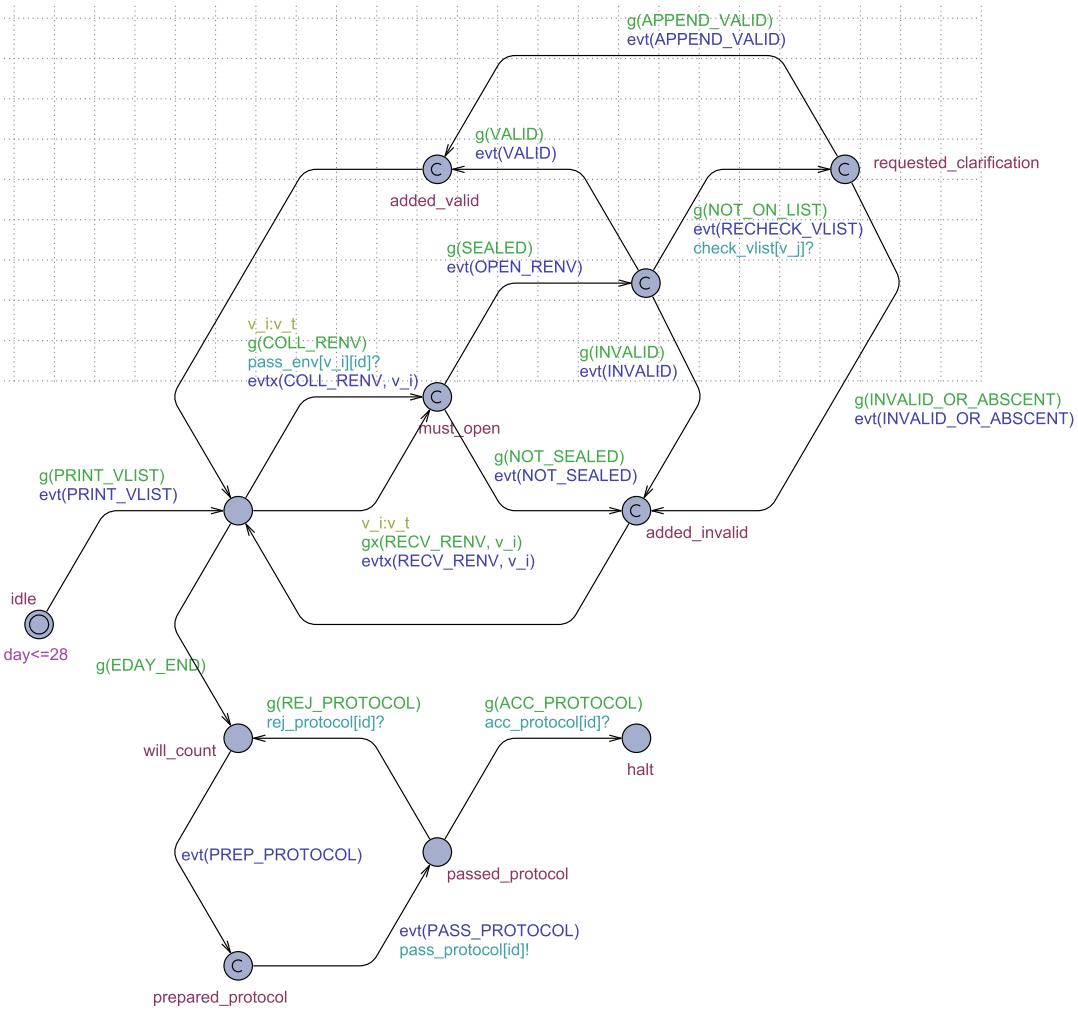


FIGURE 6.6: Electoral Commission template.

The following data structures are used within the templates:

```
typedef int[1,NC] c_t;              // candidate id
typedef int[1,NV] v_t;              // voter id
typedef int[-NMO,-1] mo_t;          // municipal office id
typedef int[-NMO-NEC,-NMO-1] ec_t;  // election commission id
typedef int[-NMO-NEC,NV] addr_t;    // domain of all agents ids

typedef struct{
    addr_t src;       // sender id
    addr_t dst;       // receiver id
    addr_t addr;      // address for EP delivery
    bool inperson;    // in-person collection preference
    v_tx pesel_of;    // PESEL number
}IntentionForm;

typedef struct{
    bool sealed;        // envelope sealed
    bool pkw_stamp;     // National Electoral Commission stamp
    bool dec_stamp;     // district electoral commission stamp
    int[0,2] cell[c_t]; // number of Xs near candidate cell
                        // (2 for any number greater than 1)
}Benv;                  // ballot envelope

typedef struct{
    addr_t src;          // sender id
    addr_t dst;          // recipient id
    Benv benv;
    mo_tx stamp;         // envelope stamp
    bool sealed;         // envelope was sealed
    bool dec_signature;  // voter's card signed
    v_tx dec_pesel;      // voter's card PESEL
}Renv;                   // return envelope

typedef struct{
    addr_t src;    // sender id
    addr_t dst;    // recipient id
    Renv renv;
}ElectionPackage;  // election package

typedef struct{
    addr_t mo_addr;  // assigned municipality
    addr_t ec_addr;  // assigned commission
    cmt_t comment;   // nominal field for
    bool changed;    // recently changed
}v_record            // an entry in the voters registry
```

Notice that the space of active agent names (or rather their unique identifiers) is partitioned with respect to the agent type. This allows to model intended parties for a given communication instance in an easy way.[5] All the variables in UPPAAL are assigned with either explicit initial or implicit default values. Thus, sometimes aliases which extend the existing data types will be used (e.g., `c_tx` for `int[0,NC]`).

[5]One of the implicit assumptions for the model is that every agent can perform only a single role.

Additionally, to improve readability and keep the graphical view 'clean' we use the following convention for the edge labels:

- all updates are of the form `evt(EVT_CODE)` or `evtx(EVT_CODE,EVT_PARAMS);`
- all guards are of the form `g(EVT_CODE)` or `gx(EVT_CODE,EVT_PARAMS).`

Where `EVT_CODE` is a string of characters (i.e., named value from the pre-defined enum of events), representing action name by means of a natural language. The sets of available/specified actions are disjoint among agents of a distinct type. On the level of the model code, functions `evt/evtx` and `g/gx` are declared using switch statement matching passed argument of `EVT_CODE` to its case clause, which should facilitate the modularity of a model and higher integrity of its graphical presentation.

6.4 Verification

To specify requirements UPPAAL admits a fragment of **CTL***, excluding the "next" and "until" modalities. Consequently, all supported formulas are of the following types: "*reachability*" (EFp), "*liveness*" (AFp), "*safety*" (EGp and AGp) and an UPPAAL specific "*leads-to*" (AG$(p \Rightarrow$ AF$p)$). Its semantics allows for non-maximal paths, in this work we will usually use EFp and AGp for specifying the desired properties.

As it was already discussed in [Jam+20b] limitations in UPPAAL requirement specification lead to its verification capabilities being rather limited. Interestingly, in the next section we will see that some of the critical requirements can still be represented by means of supported formulas, despite those limitations.

As the next step, we specify some relevant properties of the voting system by formulas of multi-agent logics, in particular the branching-time temporal logic **CTL** and the strategic logic **ATL**. Then, we transform them to a form that can be interpreted by UPPAAL, and run the model checking.

To mitigate the impact of a state-space explosion, we use abstraction on variables, originally proposed in [JK23a].

6.4.1 Specification of Properties

Following [TJR16; JKK18; Jam+20b; JKM20], we use the formulas of multi-agent logics to specify some interesting requirements on the election system. In particular, we use the branching-time temporal logic **CTL** [Eme90] and strategic logic **ATL** [AHK02]. We focus on the following requirements[6]:

(***P1***) The number of correctly received ballots cannot exceed the number of sent ballots (a weak variant of resistance against ballot stuffing);

(***P2***) For every voter, cast vote must be properly recorded and reflected by the tally (tally integrity);

(***P3***) The authorities should have no strategy to invalidate certain votes, even when the voters' preferences are known (no strategic ballot removal).

We formalize $(P1)$ and $(P2)$ by the following **CTL** formulas:

$$\varphi_{P1} \equiv \mathsf{AG}(\textstyle\sum_{i=1}^{NB} b\_received_i \leq \sum_{j=1}^{NV}(ep\_sent_j))$$
$$\varphi_{P2} \equiv \mathsf{AG}(elec\_end \wedge voted_{i,j} \Rightarrow tallied_{i,j})$$

[6] Authors would want to stress that the given properties are just an example, merely for illustration purposes, and they should not be viewed as a complete list of requirements.

where NB and NV stand for the maximum number of ballots and voters accordingly, $i \in NV$ is an arbitrarily fixed voter, and $j \in NC$ is a candidate. The $(P1)$ says that, for all possible execution paths (A) and all future time points (G), the sum of received ballots must not exceed the number of sent election packets. Similarly, $(P2)$ says that if an election is closed and a voter has cast her vote for candidate j, then it must be tallied for j.

Furthermore, we formalize $(P3)$ by the following formula of **ATL** (in fact, we formalize the negation of $(P3)$ and focus on ballot removal by a Municipal Office, thus expressing that the MO can strategically remove ballots):

$$\varphi_{\neg P3} \equiv \langle\langle MO_k \rangle\rangle \mathsf{G} \bigwedge_{i \in NV} (vreg_{i,k} \wedge pref_{i,j} \wedge elec\_end \Rightarrow \neg tallied_{i,j})$$

where $i \in NV$, $j \in NC$ and $k \in MO$ is a municipal office. The reading of $\varphi_{\neg P3}$ is: "Municipal office k has a strategy ($\langle\langle MO_k \rangle\rangle$) such that, no matter what the other agents do, at all future time points (G) for any voter i if registered in this municipality ($vreg_{i,k}$), prefers candidate j ($pref_{i,j}$) and an election is already closed ($elec\_end$), then her vote will not be tallied correctly ($\neg tallied_{i,j}$)".

6.4.2 From Agent Logics to UPPAAL Specifications

Our formalization of resistance to ballot stuffing and tally integrity has a straightforward transcription in UPPAAL specification language:

(φ_{P1}) `A[] (b_recv<=ep_sent) ,`

(φ_{P2})
```
A[] (Time.end and (Voter(i).sent_renv or Voter(i).passed_renv)
    imply recorded_link[i]==Voter(i).pref_cand) ,
```

where `b_recv`, `ep_sent` and `recorded_link` are auxiliary variables added only for the verification of related property and representing the number of received ballots, the number of sent election packages, and the mapping from voters to the way their votes have been tallied. Moreover, `pref_cand` is the voter's local variable storing her preferred candidate, `sent_renv` and `passed_renv` are labels of the corresponding locations in the voter's agent graph (see Figure 6.4).

Unfortunately, UPPAAL does not offer the verification of strategic abilities and thus does not admit **ATL** operators. To deal with that, we propose to approximate formula $\varphi_{\neg P3}$ by its *under-approximation* $\varphi^{-}_{\neg P3}$ and *over-approximation* $\varphi^{+}_{\neg P3}$, both of which are **CTL** formulas that satisfy the following conditions:

$$\begin{array}{lcl} M \models_{\mathbf{CTL}} \varphi^{-}_{\neg P3} & \Rightarrow & M \models_{\mathbf{ATL}} \varphi_{\neg P3} \\ M \models_{\mathbf{ATL}} \varphi_{\neg P3} & \Rightarrow & M \models_{\mathbf{CTL}} \varphi^{+}_{\neg P3} \end{array}$$

That is, whenever $\varphi^{-}_{\neg P3}$ is true in a model, $\varphi_{\neg P3}$ must also be true there. Moreover, if $\varphi^{+}_{\neg P3}$ is false in a model, $\varphi_{\neg P3}$ must also be false. We use the following approximations:

$$\begin{array}{lcl} \varphi^{-}_{\neg P3} & \equiv & \mathsf{AG} \bigwedge_{i \in NV} (vreg_{i,k} \wedge pref_{i,j} \wedge elec\_end \Rightarrow \neg tallied_{i,j}) \\ \varphi^{+}_{\neg P3} & \equiv & \mathsf{EG} \bigwedge_{i \in NV} (vreg_{i,k} \wedge pref_{i,j} \wedge elec\_end \Rightarrow \neg tallied_{i,j}) \end{array}$$

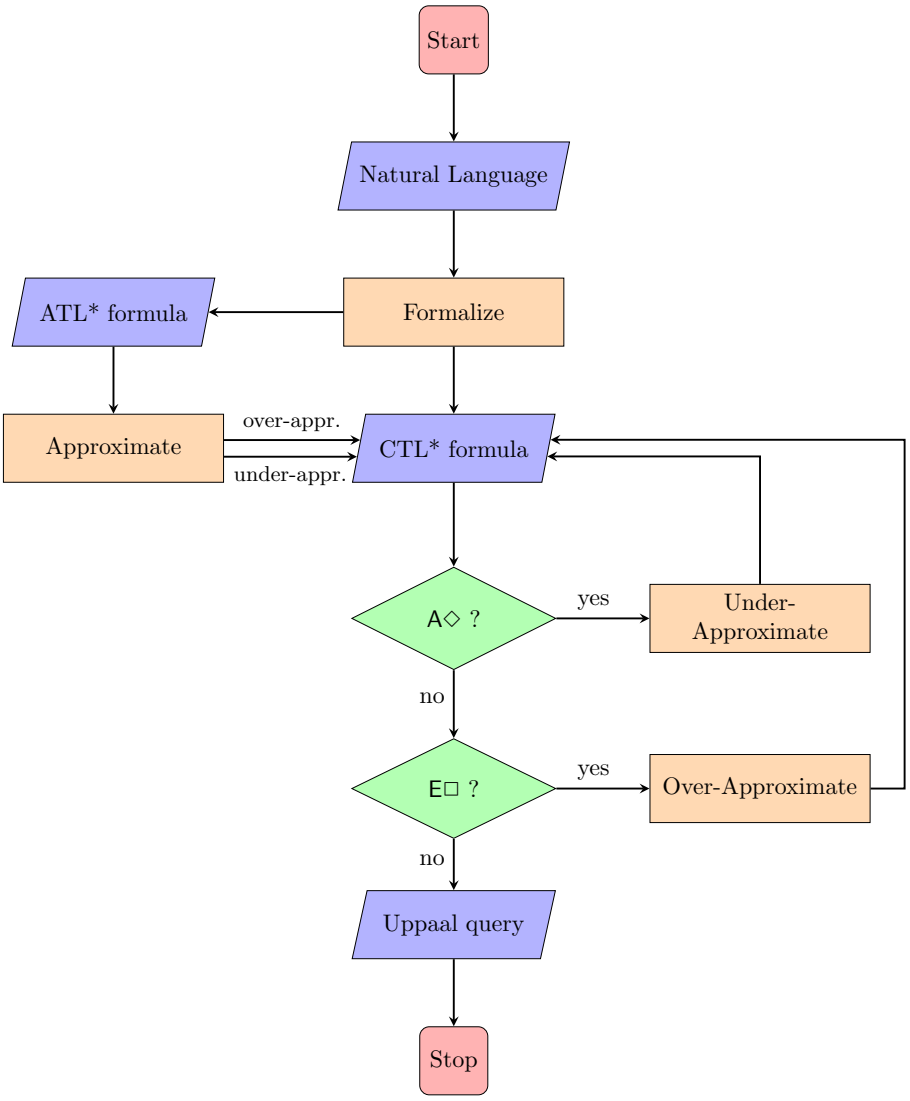


FIGURE 6.7: Flowchart illustrating the specification translation process.

This follows the intuition that, if ψ is guaranteed to always hold on all execution paths (AGψ), then it must also hold when MO plays strategically ($\langle\langle MO\rangle\rangle$G$\psi$). Moreover, if MO has a strategy to maintain ψ ($\langle\langle MO\rangle\rangle$G$\psi$), then ψ must always hold on at least one path (EGψ).

Now, formula $\varphi^{-}_{\neg P3}$ can be fed directly to UPPAAL. Unfortunately, this is not the case for the upper approximation $\varphi^{+}_{\neg P3}$, as UPPAAL does not interpret the EG combination of **CTL** operators correctly.[7] On the other hand, UPPAAL's `E[]` combination is an over-approximation of the **CTL*** EG combination. Thus, we can use it to provide "over-over-approximation" of the original specification, which finally obtains the following list of UPPAAL inputs:

($\varphi^{-}_{\neg P3}$) `A[] forall(i:v_t)(Time.end and vlist[i].mo_addr==k and vpref[i]==j imply recorded_link[i]!=j) ,`

($\varphi^{++}_{\neg P3}$) `E[] forall(i:v_t)(Time.end and vlist[i].mo_addr==k and vpref[i]==j imply recorded_link[i]!=j) ,`

[7] The satisfaction of **CTL** operators in a transition system is interpreted over *maximal runs*, i.e., ones that are either infinite or end in a state with no outgoing transitions. In contrast, UPPAAL looks at *all finite runs*. While this does not change the semantics of AG and EF, the interpretation of both AF and EG becomes nonstandard.

where `vlist` refers to the voters' registry.

Concrete model

```
typedef struct{
    bool sealed;
    bool pkw_stamp;
    bool dec_stamp;
    int[0,2] cell[c_t];
}Benv;

typedef struct{
    addr_t src;
    addr_t dst;
    Benv benv;
    mo_tx stamp;
    bool sealed;
    bool dec_signature;
    v_tx dec_pesel;
}Renv;

typedef struct{
    addr_t src;
    addr_t dst;
    Renv renv;
}ElectionPackage;
```

Abstact model

```
typedef struct{
    bool invalid;
    c_tx cell;
}Benv;

typedef struct{
    addr_t dst;
    bool invalid;
    Benv benv;
}Renv;

typedef struct{
    bool sent;
    Renv renv;
}ElectionPackage;
```

FIGURE 6.8: Fragment of code resulting from the abstraction, where all evaluations invalidating a vote are merged into a single variable (for REnv and BEnv).

6.4.3 Mitigating State Space Explosion by Abstraction of Variables

As previously discussed in Chapters 4 and 5, the phenomenon of the *state-space explosion* often obstructs model checking of the real-world systems. We recall that model checking typically involves the inspection of all possible states of the modelled system, whereas the number of such states is exponential in the number of processes and their components – in our case, the number of agent instances and their local variables [Cla+18]. This is easy to see in our experimental results for formula φ_{P1} — see Table 6.1, the part under "φ_{P1} (concrete)" — with the clear exponential growth of the verification time t and memory use m. As a consequence, the verification of φ_{P1} on the model presented in Section 6.3 scales up to only 2 voters, 1 municipal office, and 3 electoral commissions for an election with 3 candidates.

Mitigating state-space explosion has been an important topic of research for over 30 years. The most important techniques include partial-order reduction [Pel93; God96; Ger+99; Jam+20a], symbolic verification [McM93], bounded and unbounded model checking [McM02; PL03; LP07] and state/action abstraction [CC77; CGL94; GJ02]. In particular, abstraction is an intuitive model reduction method, based on the idea of clustering "similar" states of the system (so-called *concrete states*) into *abstract states*, hopefully reducing the model to a manageable size. The actual clustering must be carefully crafted. On the one hand, it must only remove information that is irrelevant for the verification of a given property, otherwise, the verification results for the abstract model will be inconclusive with respect to the original model (so-called "concrete model"). On the other hand, it has to remove sufficiently much of the concrete model, so that the model checking becomes efficient.

In this work, we use an intuitive and easy-to-use abstraction scheme, based on the removal of variables from agent graphs [JK23a]. The method allows to select subset of local variables to be removed (possibly with a subset of locations to serve as the scope of the abstraction). For example, the name of the voter's preferred candidate is irrelevant for the verification of resistance to ballot stuffing, hence the corresponding variables can be omitted in the voters' agent graphs. The abstraction generates *two* abstract models. The first one *under-approximates* the concrete model, in the sense that if formula ψ returns `true` on the abstract model, it must also be true in the concrete model. The second *over-approximates* the concrete model, i.e., if ψ returns `false` on the abstract model, it must also be false in the concrete model.

Alternatively, the user can define a mapping from the variables to a fresh variable that merges some of the information that used to be stored in the removed variables. For example, we might map the complex representation of all potential ballot faults (unsealed ballot envelope, missing stamps, more than one 'X' on the ballot, for one or more candidates) to a single boolean variable `invalid`, see Figure 6.8.

In general, these abstraction parameters should be picked firstly to reduce the number of induced states of a global model, and secondly to match the property, so that verification is conclusive. Naturally, there are plenty of ways to choose fitting parameters for a given property; it is also possible that two distinct properties are matched by the same ones, and therefore same abstract models.

6.4.4 Verification Experiments

Based on the input prepared in the previous sections, we have conducted a number of model checking experiments. All the results presented here were obtained with UPPAAL 4.1.24 (32 bit) on a laptop with Intel i7-8665U 2.11 GHz CPU, running Ubuntu 20.04 on WSL2 using 4/16 GB RAM. The outcome of the experiments is shown in Tables 6.1, 6.2 and 6.3. The notation is as follows:

- *conf* denotes the configuration of the experiment, i.e., the number of voters, Municipal Offices, Electoral Commissions, and election candidates;
- *Sat* reports the verification output, i.e., whether the model checker returned `true` or `false`;
- *t* and *m* show the time and memory used in the verification, with `memout` indicating that the model checking process ran out of memory. Note that the included time measurements could have been affected by other processes running on OS.

Table 6.1 presents the experimental results for our formalization of weak resistance to ballot stuffing. The formula has turned out to be true in all the completed verification runs. However, as already observed, the verification scales rather badly due to state-space explosion. To mitigate that, we reduced the models by abstracting away the identity of candidates and simplified the data structures representing the intention form and the election package. Moreover, we mapped variables `b_recv` and `ep_sent` to a single fresh variable `ballot_diff = ep_sent-b_recv`, so that the requirement specification became `AG(ballot_diff>=0)`. The results for model checking the under-approximating abstract model are also presented in Table 6.1.

Since the output of under-approximation was `true`, we conclude that φ_{P1} is also true in the original (concrete) model. Note that, for some configurations, the abstraction allowed to run the verification faster by orders of magnitude. Moreover, it allowed for the model checking of scenarios with 3 voters, 1 MO, 1 EC, and 3 candidates, i.e., one more voter than in the concrete case. This might seem slight, but

in some cases 3 voters are necessary to demonstrate non-trivial attacks on a voting system [ACK16].

For tally integrity (φ_{P2}), we used formula φ_{P2} proposed in Section 6.4.2. In this case, we additionally generated the under-approximating abstract models obtained by (a) mapping all the candidate names except j to a fresh value j', (b) for all voters other than i, removing their memory of the choices associated with their intention forms and election packages after they send those. The results in Table 6.2 show that the verification output was not conclusive, i.e., they do not imply whether φ_{P2} is true or false in the original model. However, the verification becomes conclusive under the assumption that voter i makes no errors and strictly follows the protocol, see the rightmost part of the table. In that case, we get `true` as the output, thus concluding the original property holds as well.

Lastly, the experimental results for strategic ballot removal ($\varphi_{\neg P3}$) are presented in Table 6.3. The table first shows the (inconclusive) output of model checking for under-approximation w.r.t. the formula, under-approximation w.r.t. the formula and the model, and over-approximation w.r.t. the formula. Thus, the original property might (but does not have to) be satisfied. Then, we fix a strategy for MO in the model, so that the municipal office sends a ballot with invalid stamps whenever the voter intends to vote for the "unwelcome" candidate.[8] The results in the rightmost part of Table 6.3 show now conclusive output: the under-approximation is true, so the original property must be true as well. Thus, the proposed strategy indeed achieves the goal specified by $\varphi_{\neg P3}$. Note that it is essential to couple matching formula- and model-related approximations, otherwise the procedure is not sound. In our case, this meant using the *under-approximating* formula $\varphi^{-}_{\neg P3}$ with the *under-approximating* abstract model of the procedure.

Despite the technical limitations on the number of voters, it was possible to discover and verify attacks violating φ_{P2} and φ_{P3} respectively. For the next step (which remains a subject of future work), we would want to scale up the model to a larger or even unbounded number of voters (the latter is currently beyond the technical feasibility of the tool) or come up with rigorous arguments that a certain number of voters is sufficient for certain cases.

6.5 Related Work

Formal verification of voting protocols has been the subject of research for over a decade. Prominent approaches include theorem proving in first-order, linear or higher order logic [PS15; BDS17; CGT18; CFL19; HGT19; HGS21], and model checking of temporal, strategic and temporal-epistemic logics [JKK18; Jam+20b; JKM20]. Most if not all results show that the task is very hard due to the prohibitive computational complexity of the underlying problems. For example, [CFL19; CGT18] conducted a formal analysis of voting protocols using ProVerif, and reported that they had to come up with workarounds for the model in order to the limitations of the tool.

Modelling and analysis of socio-technical systems are even more difficult because of the vast space of possible human behaviors, and problematic nature of the assumptions usually made about how users choose their actions. The theory of socio-technical systems dates back to the work of Trist and Bamforth in 1940s. In security, perhaps the best studied methodology is based on ceremonies [Car+12], in particular the Concertina ceremony [Bel+14; BCL15; Mar+15]. Some research has

[8] A situation underlying this scenario actually took place for some of the voters abroad [Sku20], there were also reports of this problem by voters from Lublin [Spo20] and Zakopane [Fak20].

conf	φ_{P1} (concrete)			φ_{P1} (abstract)		
	Sat	t (s)	m(MB)	Sat	t (s)	m(MB)
1,1,1,1	true	0.07	31	true	0.07	30
1,1,1,2	true	0.10	31	true	0.07	30
1,1,1,3	true	0.12	31	true	0.07	30
1,1,2,1	true	0.12	31	true	0.08	31
1,1,2,2	true	0.20	31	true	0.08	31
1,1,2,3	true	0.27	31	true	0.08	31
1,1,3,1	true	0.29	31	true	0.14	31
1,1,3,2	true	0.54	32	true	0.14	31
1,1,3,3	true	0.80	33	true	0.14	31
1,1,4,1	true	0.92	33	true	0.39	31
1,1,4,2	true	1.8	35	true	0.39	31
1,1,4,3	true	2.7	36	true	0.39	31
1,2,2,1	true	0.27	31	true	0.14	31
1,2,2,2	true	0.48	32	true	0.14	31
1,2,2,3	true	0.71	33	true	0.14	31
1,2,3,1	true	0.75	32	true	0.32	31
1,2,3,2	true	1.4	34	true	0.32	31
1,2,3,3	true	2.2	35	true	0.32	31
1,2,4,1	true	2.5	35	true	0.97	32
1,2,4,2	true	4.9	39	true	0.97	32
1,2,4,3	true	7.5	58	true	0.97	32
2,1,1,1	true	5.0	91	true	1.0	32
2,1,1,2	true	21	271	true	1.0	32
2,1,1,3	true	48	618	true	1.0	32
2,1,2,1	true	15	172	true	2.9	34
2,1,2,2	true	64	581	true	2.9	34
2,1,2,3	true	148	1332	true	2.9	34
2,1,3,1	true	52	330	true	9.5	57
2,1,3,2	true	213	1180	true	9.5	57
2,1,3,3	true	496	2796	true	9.5	57
2,1,4,1	true	190	638	true	35	83
2,1,4,2	true	789	2429	true	35	83
2,1,4,3		memout		true	35	83
2,2,2,1	true	135	990	true	18	76
2,2,2,2	true	558	3901	true	18	76
2,2,2,3		memout		true	18	76
2,2,3,1	true	445	2168	true	59	152
2,2,3,2		memout		true	59	152
2,2,3,3		memout		true	59	152
2,2,4,1		memout		true	203	350
2,2,4,2		memout		true	203	350
2,2,4,3		memout		true	203	350
3,1,1,1		memout		true	80	365
3,1,1,2		memout		true	80	365
3,1,1,3		memout		true	80	365
3,1,2,1		memout		true	241	818
3,1,2,2		memout		true	241	818
3,1,2,3		memout		true	241	818
3,1,3,1		memout		true	793	1882
3,1,3,2		memout		true	793	1882
3,1,3,3		memout		true	793	1882
3,1,4,1		memout			memout	

TABLE 6.1: Experimental results for model checking of φ_{P1}.

been also based on choreographies [Bru+21]. Moreover, game-theoretic models and analysis have been used in [BM07; JT17; Bas+17]. Here, follow up on the strand based on modelling and verification in multi-agent logics [JKK18; Jam+20b; JKM20], while trying to put more emphasis on the social part of the system outside of the voting infrastructure.

When analyzing systems that involve human agents, it is important to take into account that they behave differently from the machines and can make *errors*, or more generally, deviate from the prescribed protocol. This can happen due to a variety of reasons: misunderstanding, inattentiveness, malicious intention, or strategic self-interested action. Possible deviations from protocol in user behaviour have been

conf	φ_{P2} (concrete)			φ_{P2} (abstract)			φ_{P2} (honest abstract)		
	Sat	t (s)	m(MB)	Sat	t (s)	m(MB)	Sat	t (s)	m(MB)
1,1,1,1	false	0.07	31	false	0.06	30	true	0.05	30
1,1,1,2	false	0.09	31	false	0.06	31	true	0.06	30
1,1,1,3	false	0.11	31	false	0.07	31	true	0.06	30
1,1,2,1	false	0.12	31	false	0.08	31	true	0.05	31
1,1,2,2	false	0.23	31	false	0.10	31	true	0.07	31
1,1,2,3	false	0.25	31	false	0.13	31	true	0.08	31
1,1,3,1	false	0.31	31	false	0.15	31	true	0.12	31
1,1,3,2	false	0.52	32	false	0.25	31	true	0.13	31
1,1,3,3	false	0.75	33	false	0.35	31	true	0.17	31
1,1,4,1	false	0.90	33	false	0.40	32	true	0.19	31
1,1,4,2	false	1.7	35	false	0.76	32	true	0.33	31
1,1,4,3	false	2.6	36	false	1.1	33	true	0.69	31
1,2,2,1	false	0.26	32	false	0.14	31	true	0.09	31
1,2,2,2	false	0.45	32	false	0.22	31	true	0.13	31
1,2,2,3	false	0.66	33	false	0.31	31	true	0.16	31
1,2,3,1	false	0.70	32	false	0.33	31	true	0.16	31
1,2,3,2	false	1.4	34	false	0.61	32	true	0.26	31
1,2,3,3	false	2.0	35	false	1.3	32	true	0.36	31
1,2,4,1	false	2.3	35	false	1.0	32	true	0.40	31
1,2,4,2	false	4.7	39	false	2.0	33	true	0.97	32
1,2,4,3	false	7.1	42	false	3.0	34	true	1.1	32
2,1,1,1	false	1.8	36	false	0.95	31	true	0.28	31
2,1,1,2	false	6.9	99	false	3.0	34	true	0.72	31
2,1,1,3	false	16	220	false	6.2	54	true	1.3	32
2,1,2,1	false	7.2	79	false	2.8	34	true	0.71	31
2,1,2,2	false	30	235	false	9.2	57	true	2.3	32
2,1,2,3	false	68	535	false	20	100	true	3.4	34
2,1,3,1	false	26	137	false	9.5	41	true	2.0	33
2,1,3,2	false	104	516	false	32	114	true	5.3	36
2,1,3,3	false	239	1180	false	68	200	true	10	57
2,1,4,1	false	95	305	false	35	84	true	6.4	38
2,1,4,2	false	384	1199	false	119	233	true	18	65
2,1,4,3	false	885	2768	false	254	477	true	33	97
2,2,2,1	false	60	377	false	19	77	true	4.0	33
2,2,2,2	false	241	1439	false	61	189	true	10	53
2,2,2,3	false	549	3338	false	126	382	true	19	74
2,2,3,1	false	205	829	false	61	153	true	10	38
2,2,3,2	false	832	3318	false	199	428	true	29	81
2,2,3,3		memout		false	413	881	true	55	130
2,2,4,1	false	729	2026	false	211	338	true	33	70
2,2,4,2		memout		false	696	1075	true	92	174
2,2,4,3		memout		false	1444	2264	true	177	307
3,1,1,1	false	60	683	false	73	340	true	10	54
3,1,1,2		memout		false	384	1855	true	37	153
3,1,1,3		memout			memout		true	88	332
3,1,2,1	false	346	2387	false	224	769	true	28	99
3,1,2,2		memout			memout		true	99	293
3,1,2,3		memout			memout		true	230	667
3,1,3,1		memout		false	775	1827	true	81	201
3,1,3,2		memout			memout		true	283	639
3,1,3,3		memout			memout		true	661	1503
3,1,4,1		memout			memout		true	261	451
3,1,4,2		memout			memout		true	920	1518
3,1,4,3		memout			memout		true	2157	3626
3,2,2,1		memout			memout		true	317	667
3,2,2,2		memout			memout		true	1143	2303
3,2,2,3		memout			memout			memout	
3,2,3,1		memout			memout		true	895	1427
3,2,3,2		memout			memout			memout	
3,2,3,3		memout			memout			memout	
3,2,4,1		memout			memout		true	1065	2433
3,2,4,2		memout			memout			memout	
3,2,4,3		memout			memout			memout	
4,1,1,1		memout			memout			memout	

TABLE 6.2: Experimental results for model checking of φ_{P2}.

conf	$\varphi^{-}_{\neg P3}$ (concrete)			$\varphi^{-}_{\neg P3}$ (abstract)			$\varphi^{++}_{\neg P3}$ (concrete)			$\varphi^{-}_{\neg P3}$ (str. abstract)		
	Sat	t (s)	m(MB)	Sat	t (s)	m(MB)	Sat	t (s)	m(MB)	Sat	t (s)	m(MB)
1,1,1,1	false	0.13	31	false	0.058	30	true	0.054	31	true	0.052	30
1,1,1,2	false	0.22	31	false	0.096	31	true	0.049	31	true	0.10	30
1,1,1,3	false	0.40	32	false	0.15	31	true	0.049	31	true	0.17	31
1,1,2,1	false	0.19	31	false	0.13	31	true	0.051	31	true	0.088	31
1,1,2,2	false	0.57	32	false	0.20	31	true	0.064	31	true	0.16	31
1,1,2,3	false	1.1	34	false	0.38	31	true	0.067	31	true	0.35	31
1,1,3,1	false	0.56	32	false	0.20	31	true	0.055	32	true	0.16	31
1,1,3,2	false	1.7	35	false	0.82	31	true	0.060	32	true	0.46	31
1,1,3,3	false	3.8	39	false	1.1	32	true	0.063	32	true	0.98	32
1,1,4,1	false	1.7	34	false	0.85	32	true	0.062	32	true	0.30	31
1,1,4,2	false	6.2	42	false	1.9	34	true	0.066	32	true	1.4	33
1,1,4,3	false	14	71	false	4.0	36	true	0.067	32	true	3.3	35
1,2,2,1	false	0.48	32	false	0.22	31	true	0.061	32	true	0.15	31
1,2,2,2	false	1.6	34	false	0.54	31	true	0.077	32	true	0.40	31
1,2,2,3	false	3.3	54	false	1.0	32	true	0.064	32	true	1.2	31
1,2,3,1	false	1.4	34	false	0.46	31	true	0.087	32	true	0.26	31
1,2,3,2	false	5.1	39	false	1.5	32	true	0.076	32	true	1.1	32
1,2,3,3	false	11	81	false	3.0	34	true	0.082	32	true	2.5	33
1,2,4,1	false	4.9	37	false	1.5	33	true	0.075	32	true	0.74	32
1,2,4,2	false	18	85	false	4.9	36	true	0.075	32	true	3.5	34
1,2,4,3	false	39	142	false	10	41	true	0.075	32	true	8.4	39
2,1,1,1	false	8.3	115	false	2.0	33	true	0.065	31	true	0.62	31
2,1,1,2	false	78	833	false	23	120	true	0.076	31	true	13	77
2,1,1,3	false	323	3496	false	105	495	true	0.062	32	true	75	337
2,1,2,1	false	36	230	false	6.1	53	true	0.070	32	true	1.6	32
2,1,2,2	false	336	2145	false	71	265	true	0.088	32	true	39	152
2,1,2,3		memout		false	321	1140	true	0.079	32	true	224	748
2,1,3,1	false	105	486	false	20	84	true	0.074	32	true	4.8	36
2,1,3,2		memout		false	244	613	true	0.082	32	true	129	326
2,1,3,3		memout		false	1114	2752	true	0.10	32	true	741	1745
2,1,4,1	false	400	1160	false	75	152	true	0.083	32	true	17	48
2,1,4,2		memout		false	896	1502	true	0.094	32	true	457	761
2,1,4,3		memout			memout		true	0.11	32		memout	
2,2,2,1	false	273	1424	false	40	121	true	0.091	32	true	10	53
2,2,2,2		memout		false	443	1156	true	0.11	32	true	241	604
2,2,2,3		memout			memout		true	0.10	32	true	1332	3282
2,2,3,1	false	904	3352		memout		true	0.097	32	true	32	83
2,2,3,2		memout			memout		true	0.096	32	true	756	1388
2,2,3,3		memout			memout		true	0.094	33		memout	
2,2,4,1		memout			memout		true	0.097	33	true	107	163
2,2,4,2		memout			memout		true	0.099	33	true	2575	3482
2,2,4,3		memout			memout		true	0.14	33		memout	
3,1,1,1		memout			memout		true	0.073	32	true	92	289
3,1,1,2		memout			memout		true	0.070	32		memout	
3,1,1,3		memout			memout		true	0.083	32		memout	
3,1,2,1		memout			memout		true	0.11	32	true	167	622
3,1,2,2		memout			memout		true	0.081	32		memout	
3,1,2,3		memout			memout		true	0.089	32		memout	
3,1,3,1		memout			memout		true	0.093	32	true	296	1446
3,1,3,2		memout			memout		true	0.10	33		memout	
3,1,3,3		memout			memout		true	0.093	33		memout	
3,1,4,1		memout			memout		true	0.11	33	true	811	3828
3,1,4,2		memout			memout		true	0.14	33		memout	
3,1,4,3		memout			memout		true	0.11	33		memout	
3,2,2,1		memout			memout		true	0.14	33	true	1056	6175
3,2,2,2		memout			memout		true	0.12	33		memout	
3,2,2,3		memout			memout		true	0.11	33		memout	
3,2,3,1		memout			memout		true	0.12	33		memout	
3,2,3,2		memout			memout		true	0.15	33		memout	

TABLE 6.3: Experimental results for model checking of (under-approximating) $\varphi^{-}_{\neg P3}$ and $\varphi^{++}_{\neg P3}$ in a model, where MO may prepare EP without a valid stamp.

studied in [BB06; BRS16], and we follow up on those ideas. To this end, we use the *skilled-human approach* to capture a variety of users' behaviours in our model of postal voting. That is, we extend the protocol specification of an "honest" behaviour through a hierarchy of *deviation sets*, i.e., sets of actions that deviate from the protocol and expand the repertoire of the participants. As pointed out in [BRS16], there is a trade-off between the breadth of the deviation model and the computational feasibility of the formal analysis. In fact, our experiments in Section 6.4 show that even for the skilled human approach only, the explicit state model checking becomes hard enough, and dedicated techniques must be used to mitigate the complexity.

Related verification tools. We use the UPPAAL model checker [BDL04] in our case study, mainly because of its GUI and flexible system specification language. Other verification tools that we considered when preparing the study are:

- *MCMAS* [LQR17]: a state-of-art OBDD-based symbolic model checker for agent-based systems. The system is described using ISPL (Interpreted Systems Programming Language), and the requirements are specified as formulae of strategic or temporal-epistemic properties;
- *Tamarin-prover* [Mei+13]: a tool for security protocol verification and not a model checker per se. The system specification language is based on multiset rewriting theories and the requirements are specified as first-order temporal properties;
- *STV* [KJK19; Kur+21]: an experimental toolbox for explicit-state model checking of strategic properties; at the moment, custom input models are not fully supported and may lack documentation;
- *ProVerif* [Bla+16]: an automated cryptographic protocol verifier, in the symbolic (Dolev-Yao) model. The protocol representation is based on Horn clauses; it can be used for proving secrecy, authentication and equivalence properties.

Among the above tools, only STV, UPPAAL, and (to a lesser extent) TAMARIN provide a graphical view of the system structure. Of those, only UPPAAL allows for *interactive graphical system specification*, which we claim to be crucial in modelling and analysis of voting protocols. Real-life voting procedures include the interaction of numerous participants, each of them with a possibly different agenda and capabilities. Furthermore, the behaviour of most participants is characterized with a mixture of controllable and uncontrollable non-determinism. In consequence, interactive GUI is crucial if we want to ensure that the model we verify and the one we want to verify are the same thing, cf. [Jam+20b] for discussion.

Moreover, UPPAAL (and, to a smaller extent, STV) allow for parameterized specification of the system, without forcing the designer to program a dedicated model-generator (e.g., as in the verification of SELENE protocol with MCMAS in [JKK18]).

6.6 Conclusions

In this chapter, we demonstrates how multi-agent methodology can be used to specify and analyse the impact of human aspects on the security and integrity of voting protocols. We also argue that postal voting protocols provide good material for case studies, that will hopefully increase our understanding of the subject, and help to design better protocols.

Speaking in more concrete terms, we propose a preliminary analysis of the Polish postal vote used in the presidential election of 2020. We use Multi-Agent Graphs to represent the participants and their interaction, and formulas of multi-agent logics **CTL** and **ATL** to encode interesting properties. Then, we transform those to match the

input of the state-of-the-art model checker UPPAAL. This way, the obtained models are given an intuitive visual representation and a modular structure that allows for easier modifications and detection of errors. We also use a recently proposed method of state abstraction by variable removal to reduce the models and mitigate state-space explosion.

Despite the limitations of UPPAAL in the expressive power of its property specification language, we have managed to conclusively verify the selected properties for non-trivial configurations of voters, electoral commissions, and candidates. To this end, we used approximation over formulas (by providing weaker or stronger versions of the original requirements) and approximation w.r.t. models (by generating appropriate abstract models). Choosing the right approximations was by no means obvious and required some skill. We believe, however, that this is inevitable: successful formal analysis of real-life scenarios requires both science and art. With more than a little bit of understanding and domain knowledge.

For future work we plan to employ an alternative verification tool to conduct analysis of a broader scope of interesting properties and adapt our abstraction methods for that. One of the interesting directions is to adopt a methodology defining families of possible human mistakes as in [SV20], where the human agent deviates from the protocol through a combination of skipping, modifying, or adding action(s), or in [BGS22], which also advocates using the epistemic modal logic distinguishing between knowledge and possession.

Chapter 7

One Model to Bring Them All: Hierarchical and Parameterized Specification of Polish Postal Voting

As was already observed in Chapter 3 and Chapter 6, the complexity of multi-agent system verification in practice often quickly gets obstructed by the state space explosion, and sometimes even in basic configurations with a few agents only. Consequently, this necessitates the use of model reduction techniques, such as the abstraction method. On the other hand, both of the aforementioned parameterized models provided an arguably high-level (coarse) view, which omits a lot of details, and mainly focused on capturing the honest behaviour of the agents involved, with exception to some experimental cases, when the model was extended to accommodate verification of particular properties.

In this chapter, we present a more sophisticated approach of modelling, which aims to refine the models of Polish Postal Voting 2020 from Chapter 6 while maintaining clarity and simplicity, both visual and procedural. We propose a preliminary hierarchical model of the procedure in the form of a Multi-Agent Graph and formalize selected integrity and security properties by formulas of agent logic. Within a single parameterized specification of a model we capture a whole family of model variants, which can be easily selected for verification, both in whole and in parts. The proposed modelling approach is intuitive, scalable and highly flexible, and provides a clear distinction between knowing some abstract facts and possessing some physical object, which can be queried for facts about its attributes and possibly interacted with to alter them. Then, we transform the models and formulas so that they can be input to the state-of-art model checker UPPAAL. Furthermore, we discuss how certain internal relations between model variants can be derived and used to propagate the verification result of one further onto the other.

7.1 Introduction

Postal Voting (PV) was the only way for voting in Polish Presidential Elections (PPE) 2020 for voters abroad, under quarantine, on ships and in a few other designated areas, and an optional way locally. The whole procedure was organized under conditions of both political and time pressure, with no proper analysis or prior study of recently proposed schemes (e.g., [BRT13; KS19]). This only served to increase the number of vulnerabilities and issues, that could have arguably been avoided otherwise [And20]. For example, many voters reported receiving invalid ballots, which were not genuinely stamped: these were merely photocopied by their local authority unit, either deliberately or due to a lack of clear guidelines to follow [Sku20; Fak20; Spo20].

In this chapter we will solely focus on the postal part of elections. According to [Pańd], during the second round of PPE 2020 4.81% of votes were cast using postal channel, of which 64.98% were sent abroad. Although those votes make up a relatively small fraction of the total, an exploit of PV could be sufficient to change the actual result. Note that the last three presidential elections in Poland all ended in the second round and with a fairly narrow margin-of-victory: 6.02% in 2010, 3.1% in 2015, and 2.06% in 2020 [Pańb; Pańc; Pańd].

7.2 Voting Scenario

This section provides a brief overview of the Postal Voting (PV) protocol used in the Polish Presidential Elections 2020 (PPE). The description largely overlaps with that in Section 6.2 and is included here for the sake of completeness. The complete list of associated legislations (including notices, resolutions and ordinances), public data sheets of results and more, was aggregated by the National Electoral Commission and made available online [Pańd]. Additionally, the comprehensive list of terminology (used in legal acts) together with Polish-to-English translations can be found in [Głó].

In general terms, the scheme runs as follows:

- A Voter who intends to vote by post must fill an Intention Form (IF) with personal information and address for delivery of Election Package (EP), and declare that with the Municipal Office (or Consulate), to which she is assigned in the voters register. It is also possible to collect EP in-person by specifying that in-place of delivery address.
- Municipal Office (or Consulate) collects the intentions, checks the voters register, and prepares and distributes EPs to eligible applicants. A complete EP contains an instruction card, a stamped ballot, a voting card, and two envelopes: one for the ballot and one to be returned.
- Upon receipt of EP, a voter should put a single 'X' against a candidate of her choice on the ballot and write down her national identifier number (PESEL) on a voting card and sign that. A ballot must be put into a ballot envelope (BE), which is put together with a voting card inside a return envelope (RE). Both BE and RE must be sealed. Deviation in any of the above steps (e.g., misplacement of a voting card) would result with invalidated vote.
- A completed RE can either be turned in directly to the Election Commission or sent by post to the Municipal Office (or Consulate), where it will be stored until the day of elections.

- During the day of elections, an Election Commission processes received REs, checking information on the voting card with an electoral registry (i.e., that signature is present and PESEL belongs to an eligible voter). If all checks are passed, then BE is put into the ballot box.
- Finally, when all REs were collected and processed, the BEs from the ballot box are opened and the results are tallied.

According to Electoral Code [Pańa], the candidate who receives more than half of all validly cast votes becomes a President. If none of the candidates received the necessary number of votes, in 14 days a second round of elections is held between the two candidates with the highest number of votes.

7.3 Model

This section will outline the concepts of the hierarchical modelling approach and demonstrate its application to the case of Polish Postal Voting using UPPAAL.

7.3.1 Parametrisation through Configuration

We define two kinds of configuration: *static* for the MAS (denoted as $\mathcal{C}^*$) and *dynamic* for the agents (denoted as $\mathcal{C}_i$ for agent i). The former mainly concerns the predefined constants, variable domains and their initial values. For example: a number of candidates, the size of a tally, number of choices on the ballot. Whereas, the latter considers what kind of agent *activity* (a collection of compound actions) will be taken into account and whether those could repeat. For example: if the Voter forgets to sign a voting card, trades her election package to another agent, or attempts to vote more than once.

An agent activity can either be *invoked* by a parent agent process or *independent* from others (i.e., continuously running in the background). The former provides a simple control flow structure, while the latter gives a realistic representation of procedures that can take place at virtually any time. An agent template with its activities will be called a *module*.

Intuitively dynamic configuration space can be viewed as a lattice (obtained from the powerset of captured actions with subset inclusion). A simplified example – only capturing abstract activities – is depicted in Figure 7.2. The top-/bottom-most dynamic configurations within a module will be referred to as apex-/baseline-configuration respectively.

Note that, due to dynamic being multi-dimensional (one per agent type), identification of an anti-chain, which for a given safety properties "splits" the space into configurations that satisfy it and those that do not, becomes non-trivial.

From a given configuration, an input XML model for UPPAAL is generated using a template engine. It allows extending the built-in syntax of UPPAAL by injecting code fragments into the model specification, which shall be pre-computed against the configuration data. For instance, the constant number of candidates `NC=2` parameterized the variable domains for the tally (`tally[NC]`) and crossed cell on the ballot (`ep_x`) as `Array<int>(2)` and `Range<-1,2>` respectively. In this way, by writing a single specification file, we effectively obtain a whole family of model variants.

The source file for generating the model together with its configurations used in experiments can be found in https://github.com/submission29/model.

```
{
    "NV":1,         // Number of Voters
    "NC":2,         // Number of Candidates
    "NA":1,         // Number of Authority Districts
    "NPO":1,        // Number of Postal Offices
    "NEP":2,        // Number of Election Packages
    "NIF":2         // Number of Intention Forms
}
```

FIGURE 7.1: An example static-configuration (a JSON-file).

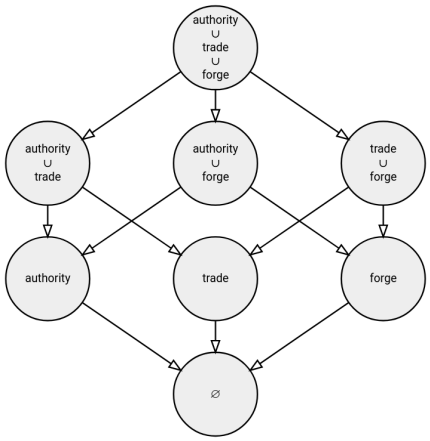


FIGURE 7.2: Hasse diagram of the fragment of Voter's dynamic configuration space, related to acquiring an election package, and with actions from the same activities collapsed. Nodes stand for possible activity and directed edges for the proper superset relation.

7.3.2 Active Agents and Passive Objects

The voter module (Fig. 7.3) serves as an entry point and provides an intuitive guideline of possible interweaving of the agent's activities (Figures 7.4, 7.6 and 7.7a), such as:

- `Voter__intention`: declaration of an intention to vote by post,
- `Voter__acquire`: collection of an election package,
- `Voter__fill`: filling up the owned election package,
- `Voter__handout`: hand out the election package to another agent,
- `Voter__cast`: casting the vote by sending the return envelope from the election package.

An authority module represents the combined entity of Municipal Office and Election Commission employees. To accurately capture the nature of its distributed activities, those were modelled as independent subprocesses, such as:

- `Authority__intention`: collection and storage of the intentions from the Voters,
- `Authority__prepare`: its processing and preparing of the election packages,
- `Authority__distribute`: distribution of election packages to the Voters,
- `Authority__aggregate`: collection and storage of return envelopes,
- `Authority__record`: validation of cast votes and tallying the results.

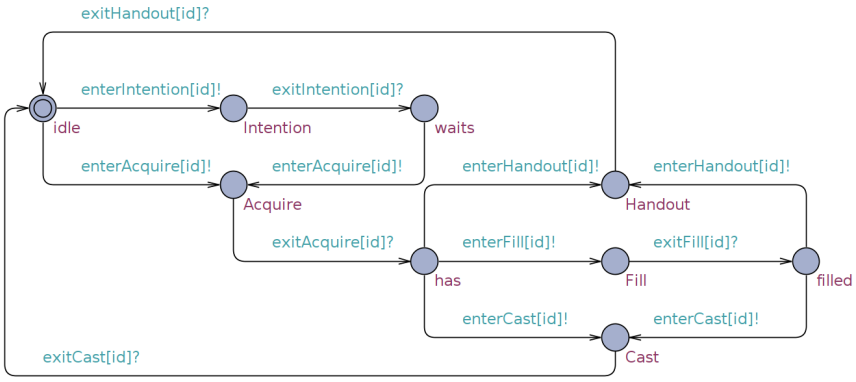


FIGURE 7.3: A Voter agent graph template, which serves as an entry-point for a module instance.

Lastly, the Postal Office module includes three independently executed activities:

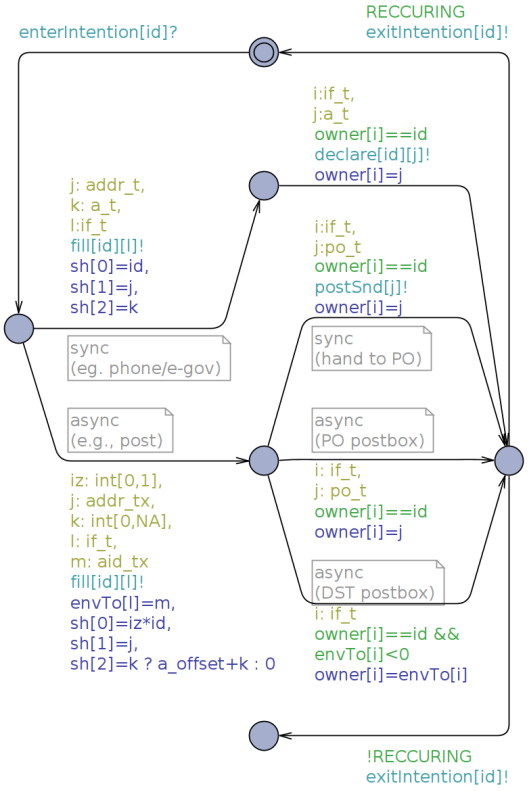
- `PostOffice__accept`: pick up of the package from the sender (or postoffice letterbox),
- `PostOffice__deliver`: forward the package to its receiver (either directly or through a letterbox),
- `PostOffice__alter`: disclosure of package content and/or its modification.

Additionally, physical objects are also modelled as processes, so that the difference between the knowledge (captured by variables) and possession of an actual object, attributes of which can be queried or changed only by its owner, is more distinctive. Formally speaking, those can also be classified as agents, however reasoning about them in such a way makes little sense and could lead to a variety of semantic paradoxes (e.g., an object being a part of coalition or a having a strategy). Therefore, one can assume that objects – despite being represented as a set of processes – are an auxiliary part of the *environment*.

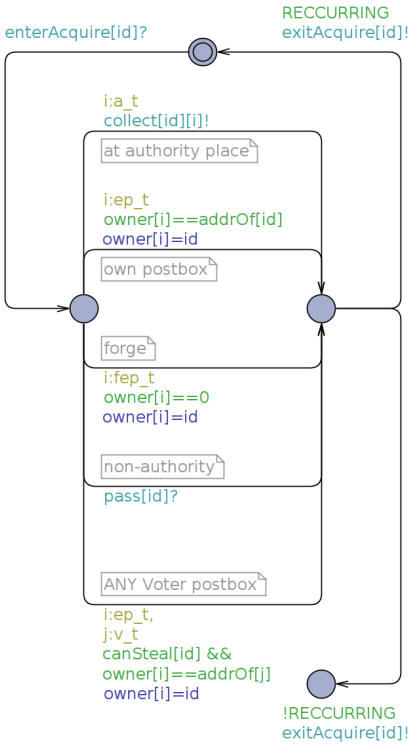
For this model we utilize two kinds of objects: Intention Form (IF) and Election Package (EP). When modelling objects via processes, the upper bound on the number of their instances must be set. Here, as the fairness assumption, we also fix the lower bound equal to the number of Voters, such that there is at least one IF and EP per Voter. IF is composed of the voter's signature, the address for EP delivery and the target MO. EP is represented as a tuple of ballot stamp, voter's card signature, and selected candidate on the ballot (with -1 for the case, when there is more than a single cross). A filled EP with sealed envelopes corresponds to a Return Envelope (RE).

In all aforementioned cases, processes instantiated from the same module will share the same identifiers and variables. Each instance of the agent module (resp. object) is assigned an identifier that is unique within other modules (resp. objects). Moreover, the space of values is partitioned in a way that allows to derive from an ID its agent type. Additionally, every agent ID is mapped to a secondary ID representing its mailbox or storage. This allows capturing the handover of objects in a rather elegant way (e.g., in contrast with merely transmitting the object's data through traditional buffers) – passing "ownership" to a receiver's secondary ID.

For every invoked activity a pair of synchronous transitions specifying its start and finish must be defined in (invoked) activity and (invoking) parent templates. By convention, channel names for such transition are built from prefixing an activity



(A) When intention is expressed synchronously it must be complete (i.e., no field may be left blank). This is not enforced for the asynchronous alternative. It is assumed that Voter "knows" her national ID number and signature, and addresses of all other agents.



(B) An EP can be collected from Authority, sent to Voter's mailbox, forged, traded or stolen from other Voter's mailbox. By assumption, forged EPs do not have a valid stamp.

FIGURE 7.4: Template for invoked activities Voter__intention (A) and Voter__acquire (B).

name with `enter` for the start and `exit` for the finish, possibly followed by an array index operator `[]` to confine the synchronisation to processes of the same module instance only (i.e., of the same ID). For example, Voter's activity `Intention` (Fig. 7.4a) is invoked from a parental process of the Voter (Figure 7.3) by taking a pair of *matching*[1] transitions labelled with `enterIntention[id]?` and `enterIntention[id]!`.

7.4 Verification

We will now specify some voting system properties and demonstrate, how those can be verified or at least approximated: It should be emphasised that these requirements do not make up an exhaustive list and mainly include properties, which authors found interesting for illustration.

Experimental results will be mainly around a static-configuration with single Voter as in Figure 7.1. Already when increasing the number of voters up to 2, the verification aborts due to lack of memory. The reason for this was a so-called State-Space Explosion (SSE) problem – an exponential blow-up in the number of states with relation to the number of processes and variables. SSE is known as one of the

[1]Note, that for all occurring templates, the variable `id` is a template parameter (i.e., it is context-sensitive). Upon process instantiation every occurrence of `id` shall be substituted with a constant literal.

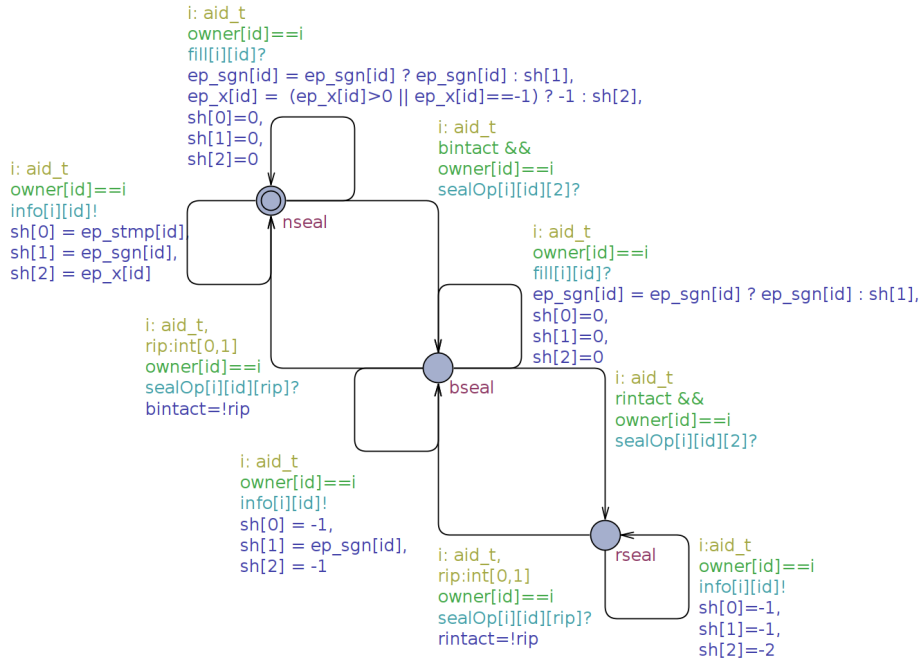


FIGURE 7.5: Election package template. A voting card can only be filled in `nseal` and `bseal` location, and a ballot only in `nseal`. By assumption, these forms cannot be easily duplicated, therefore only type of possible "overwriting" existing attribute is to put extra cross on the ballot.

major obstacles for practical model checking of real-world systems, and a variety of techniques to mitigate that were proposed in the last 30 years. Notable examples, that are compatible with MAS semantics, are: partial-order reduction [Pel93; Jam+20a], on-the-fly verification [CKM01] and state abstraction [CC77; JK23a].

For reachability properties: when satisfied, the witnessing path will usually propagate to more complex configurations. Whereas in case of the satisfied safety properties, despite lack of formal guarantees the same would hold for other static configurations, especially those with a greater number of voters, obtained results could greatly increase the confidence in the system. Moreover, it has been known that for some properties, it often suffices to look for small counter-examples [ACK16].

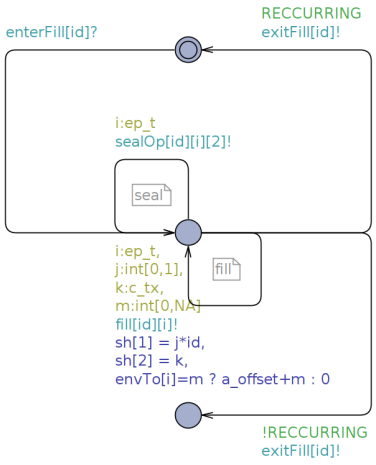
Note that, if a system — where some static-configuration is fixed and all modules are in their apex-configuration — satisfies some safety property, then it is naturally guaranteed to hold for every other dynamic-configuration as well, as those will always have less or equal number of possible transitions and reachable states.

We start with a weak resistance to ballot stuffing, saying that the number of tallied votes altogether must never exceed the number of eligible voters. It is expressed by:

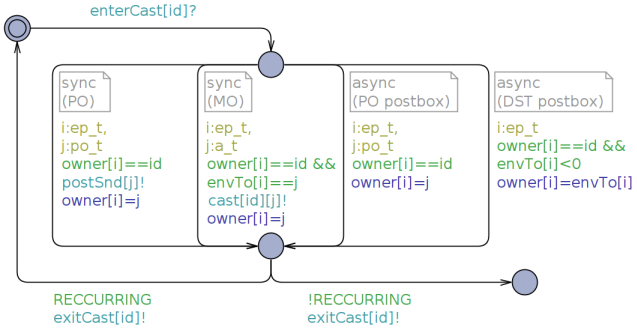
$$\texttt{AG(NV >= sum(i:int[1,NC])tally[i])} \qquad (\varphi_1)$$

where `NV` and `NC` are the total number of voters and candidates from the static-configuration. The verification of a model over selected configurations all resulted with "property satisfied" output, when the amount of required memory did not exceed the available one.

Next, consider the disenfranchisement attack, which deprives an eligible voter an opportunity to to cast a valid vote. For simplicity, we use a formula that captures an



(A) An EP can be manipulated in two ways: sealing the ballot or return envelope, and filling the forms, which are not inside a sealed envelope yet.



(B) A completed RE can either be personally placed into Municipal Office postbox, turned in to Election Commission, or handed in for delivery by the Post Office.

FIGURE 7.6: Template for invoked activities Voter__fill (A) and Voter__cast (B).

static configuration	#St	t (sec)	φ_1	φ_2
1,1,1,1,1,1	7.9e+5	5.3	$\oplus$	$\odot, \ominus$
1,1,1,1,1,2	7.8e+6	102	$\oplus$	$\odot, \ominus$
1,1,1,1,2,1	9.1e+6	91	$\oplus$	$\odot, \ominus$
1,1,1,1,2,2	2.8e+7	466	$\oplus$	$\odot, \ominus$
1,2,1,1,2,2	3.9e+7	508	$\oplus$	$\odot, \ominus$
1,2,1,2,2,2	memout		$\odot$	$\odot, \ominus$
2,1,1,1,2,2	memout		$\odot$	$\odot, \ominus$
2,2,2,2,2,2	memout		$\odot$	$\odot, \ominus$

TABLE 7.1: Experimental results for selected static-configurations.

attack against all voters:

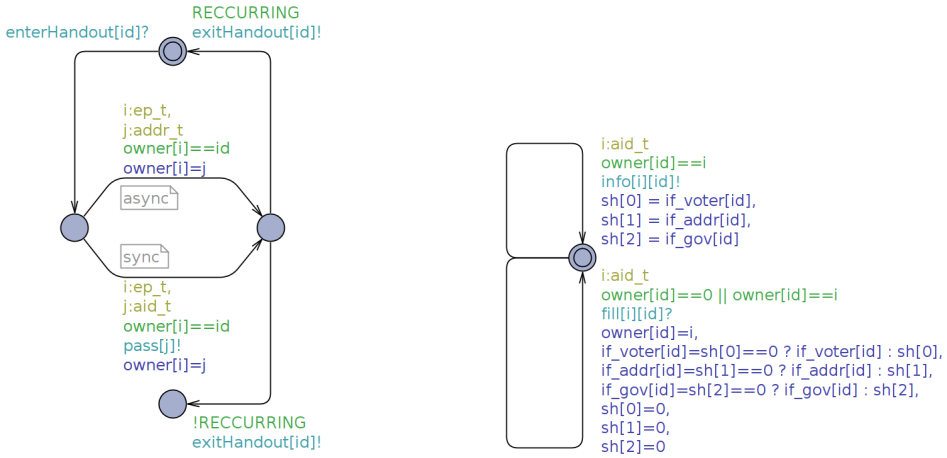
$$\texttt{AG(0==sum(i:int[1,NC])tally[i])} \qquad (\varphi_2)$$

That can be deployed by: (1) Authority with fixed a dynamic-configuration (partial strategy), such that target Voter(s) never receives an EP or gets sent one with no stamp, or (2) Postal Office, which "loses" or "alters" certain EP and RE. Notably, there is no way the Voter could defend against the former, while in the latter case, a Voter could protect themselves by only communicating with the Authority directly.

7.5 Experimental Results

We report a series of experimental results with different configurations in Table 7.1. Those were performed using Uppaal v4.1.24 on a machine with Intel i7-8665U 2.11 GHz CPU with 16 GB RAM, running Ubuntu 22.04.

In Table 7.1 columns show the static-configuration (in order: number of voters, candidates, authority entities, postal offices, election packages and intention forms), number of states in the unfolded LTS for the apex dynamic-configuration with no



(A) For the handout EP can either be put into the receiver's postbox (async) or turned in personally (sync).

(B) By assumption, anyone can print a blank IF (and therefore claim an unowned instance of that). Moreover, an "overwrite" of its fields can be accomplished by using an ordinary photocopy with a small piece of paper overlaying old values.

FIGURE 7.7: Template for invoked activity Voter__handout (A) and Intention Form (B).

reoccurring activities[2], time spent for its generation and verification. The last two columns display the verification output for φ_1 and φ_2, where $\oplus$, $\odot$, $\ominus$ say that the safety property: holds for the apex-configuration (i.e., all its descendant as well), holds for some dynamic-configuration, and does not hold for some dynamic-configurations respectively. Note that for $\odot$, $\ominus$ the number of states and verification time might differ (it can only be less or equal than that from the "ancestor").

7.6 Related Work

The work also draws inspiration from analysis of Swiss Postal Voting Process [KS19], where authors follow Risk Assessment methodology, but report a rather abstract model, and perform the whole analysis manually.

For the hierarchical representation of multi-agent system behaviour we utilize MAS graphs proposed in [JRK22]. The parent-child relation is established by means of designated synchronous transitions. Moreover, inspired by fundamental ideas from [Hoa13], the system scalability and modularity are further extended by the isolation of constants, static and dynamic behaviour parameters.

Similarly to [BGS22], we design a model that captures the distinction between the knowledge and the possession of an actual object. However, our approach to capturing human behaviour is more general: deviations from "honest behaviour" are not restricted solely to those arising from the exchange of information or interaction with the objects. Moreover, we propose a much broader family of behavioural categories, which is defined by the number of agent actions and activities (and not fixed to a powerset over 4 predefined traits of ceremony participants). It extends the top-down modelling paradigm of "skilled human" from [BRS16] by adding parameterized configurations and having modular components. Our approach for behavioural categories is also more general: firstly, it is multidimensional (i.e., defined

[2]Due to the SSE, we restrict the scope of experiments to the sub-lattice of the product of all dynamic-configurations, where each activity of an agent occurs at most once.

for each agent), and, secondly, it is not classified wrt fixed semantics (i.e., not limited to merely partitioning behaviours into honest and dishonest).

7.7 Conclusions

In this chapter, we proposed a hierarchical modelling approach and demonstrated that with a model of postal voting protocol. We show how various socio-technical interactions could be captured while maintaining the system specification clear and "readable". Despite Uppaal's limited expressive power, it can be used for verification of safety properties or witness checks for the success of a partial strategy (captured by certain dynamic-configuration for the agent modules in coalition). We do believe the modelling method itself is generic enough to be applied beyond the PPV scenario.

While we were able to manually follow the hierarchical paradigm throughout designing the model, it would be useful to have an assistance tool to guarantee coherence with the proposed approach.

For future work, we plan to combine the verification workflow with reduction techniques and conduct the analysis with even more complex configurations and scrupulous interactions. This could also involve utilisation of other verification tools, which support more expressive logics [KJK19; Mei+13].

Chapter 8

Scalable Verification of Social Explainable AI by Variable Abstraction

Social Explainable AI is a new direction in artificial intelligence that emphasises decentralisation, transparency, social context, and focus on human users. SAI research is still at an early stage and concentrates mainly on delivering the intended functionalities. At the same time, formal analysis and verification of the proposed solutions is rare. The experimental results presented in Chapter 5 examine one specific variant of gossip learning for the Social AI (SAI) protocol.

In this chapter, we will provide a more comprehensive study of the parameterized family of SAI models, which involves different types of network topologies, attackers and information exchange. In particular, we demonstrate how branching-time logic formulas can be used to express certain interesting properties of the protocol, and then verified using the state-of-the-art temporal model checker UPPAAL. In cases like this, the state-space explosion and the resulting complexity of verification is a significant problem. We mitigate this through state abstraction and demonstrate the advantages in practice by using a novel tool for user-friendly abstractions EASYABSTRACT from Chapter 5.

8.1 Introduction

Artificial intelligence solutions have become ubiquitous in daily life, including social media, car navigation, recommendation algorithms, etc. Moreover, AI provides back-end solutions to many business processes, resulting in a huge societal and economic

impact. *Social Explainable AI (SAI)* is a new, powerful idea in artificial intelligence research [Soc24; CKO22]. SAI emphasises decentralisation, human-centricity, and explainability, which is in line with the trend to move away from classical, centralised machine learning. This is essential – not only for technical reasons like scalability, but also to meet the more and more stringent ethical requirements with respect to transparency and trustworthiness of data storage and computation [Dra+20; Ott+22]. Even more importantly, SAI tries to put the human user in the spotlight, and move the focus away from the technological infrastructure [CP18; Top+21; FPC22].

Social Explainable AI is a new concept and a subject of ongoing research. It remains to be seen if it will deliver effective, transparent, and mindful AI solutions. SAI should be extensively studied, including formal verification of relevant requirements. Importantly, this should encompass the possible side effects of interaction that involves AI components and human users in complex environments. In particular, one should carefully analyse the possibilities of adversarial misuse and abuse of the interaction, e.g., by means of impersonation or man-in-the-middle attacks [DY83; Gol11]. In those scenarios, one or more nodes of the interaction network are taken over by a malicious party that tries to disrupt communication, corrupt data, and/or spread false information. The design of SAI must be resistant to such abuse; otherwise it contains a vulnerability which will be sooner or later exploited. While the topic of adversarial attacks on machine learning algorithms is an established topic of research [GMP18; KW19; Kum+20], the research on SAI has mainly focused on its expected functionalities and ideal environments of execution.[1] This is probably because SAI environments are very complex: both conceptually, computationally, and socially. Thus, a realistic study of their possible *unintended* behaviors is very challenging.

[KJS23] proposed that SAI can benefit from the use of formal methods to analyze the behaviours that can possibly emerge. In particular, a SAI protocol can be seen as an example of a *multi-agent system* [Wei99; SL09] that includes human as well as artificial agents interacting in a mixed social/computational environment. Consequently, one can use *model checking* [Cla+18], which is arguably the most successful framework of *formal verification*, to specify, visualise, and analyse SAI designs with respect to the relevant properties. The study in [KJS23] concentrated on the verification of properties related to *strategic ability* of agents and their groups to achieve their goals [BGJ15], using appropriate model checking tools, such as STV [Kur+21].

The results were promising but also showed that the high computational complexity of verification for strategic properties only allows for the analysis of very simple models. In this work, we propose to focus on *temporal* instead of strategic model checking. This way, we lose some of the expressivity with respect to which requirements can be analysed, but we gain on the feasibility of the verification process. We use *multi-agent graphs* [JK23a] to specify the agents and their interaction, and formulas of *branching-time temporal logic* **CTL** [Eme90] to formalize the interesting properties. Further, we apply the state-of-the-art model checker UPPAAL [BDL04] to automatically verify those properties. Despite lower verification complexity, the formal models of SAI still suffer from the so-called *state space explosion* [Cla+18]. To mitigate it, we use the recent experimental model reduction tool EASYABSTRACT [JK23b] that clusters similar states of the formal model in a provably correct and user-friendly way.

[1] With the notable exception of [KJS23].

8.2 Social Explainable AI

Social Explainable AI [Soc24; CKO22; FPC22], *SAI* in short, is a powerful idea whose goal is to address important drawbacks of the currently dominant AI approaches. First and foremost, the current machine learning-based systems are predominantly centralised. The huge size of data collections used in the learning process, as well as the complexity of the resulting AI models (typically, deep neural networks), make the resulting AI systems effectively black boxes, i.e., systems that do their job remarkably well, but resist deeper interpretation by users and even by machine learning experts. This naturally raises issues of safety and trustworthiness. Moreover, that often requires to store a large collection of sensitive data in a single, central location, which in turn raises the questions of feasibility, privacy, data protection, as well as compliance with legal regulations regarding data ownership.

In contrast, SAI envisions novel machine learning-based AI systems with the following foci:

Individuation. The main architecture is based on "Personal AI Valets" (PAIVs) associated with human users, and each acting as the user's proxy in a complex ecosystem of interacting agents;

Personalisation. Each PAIV processes the data through an explainable AI model tailored to the specific characteristics of its user;

Purposeful interaction. The machine learning and decision making in PAIVs are obtained through interaction, starting from the local AI models and making them interact with each other;

Human-centricity. AI algorithms and PAIV interactions are driven by quantifiable models of the individual and social behaviour of their human users;

Explainability by design. Machine Learning techniques produce explainable AI models through quantifiable human behavioural models and network science analysis.

The current attempts at building SAI [Pal+23a; Pal+23b] use *gossip learning* as the ML regime for PAIVs [Soc22; HDJ19; HDJ21]. An experimental simulation tool to assess the effectiveness of the process and functionality of the resulting AI components is available in [LBP22]. In this work, we focus on *modeling the multi-agent interaction* in the learning process, and *formal verification of the interaction* by model checking. We model the network of PAIVs as an *asynchronous multi-agent graph* [JK23a], *MAS graph* in short, and formalize its properties as formulas of *branching-time temporal logic* **CTL** *[Eme90]*. Then, we use the state-of-art model checker UPPAAL [BDL04] to verify interesting properties of SAI, and the recent experimental model reduction tool EASYABSTRACT [JK23b] to mitigate the complexity of the verification process.

The formal framework is introduced in Section 8.3. In Section 8.4, we present our MAS graphs for SAI, including models of possible adversarial behaviors, inspired by [KJS23]. In Section 8.5, we formalize several properties and conduct model checking experiments.

8.3 Formal Framework

We will now briefly recall the formal machinery from Chapters 2 and 4 used in the rest of the chapter. For more details and in-depth discussions, we refer the interested reader to [Eme90; JK23a; JK23b].

To specify the possible behaviours of the system, we use *MAS graphs* [JK23a], based on standard models of concurrency [Pri83], and compatible with UPPAAL model specifications [BDL04]. They are composed of the *agent graphs*. A *MAS template* treats each agent graph as a template and specifies the number of its instances that occur in the verified system (each differing only by the value of the special variable `id`).

An example of MAS template, which is parameterized by a variable `id`, is shown in Fig. 8.1.

Every MAS graph can be transformed to its *combined MAS graph* representing the behaviour of the system as a whole. A *global model* is obtained from a combined MAS graph by unfolding it to the labelled transition system where states are defined by a tuple of location and valuations of all the variables. Such models are usually huge due to the well-known state-space explosion. Very often, this is the main bottleneck of the verification procedure.

Branching-time logic ACTL. To express requirements, we use the *universal fragment of the branching-time logic* **CTL** [Eme90], denoted **ACTL**, with A ("for every path") as the only path quantifier. Refer to Definition 2.8 and Definition 2.9 for details on its syntax and semantics.

Recall that UPPAAL uses a non-standard interpretation of formulas using the AF combination of quantifiers, as it admits non-maximal runs in the interpretation of "for every path" (cf, Section 2.4). Fortunately, we have come up with a fix that restores the standard semantics. We present it in Section 8.5.

User-friendly state abstraction. To mitigate the impact of state space explosion, we use *state abstraction*, i.e., a method that reduces the state space by clustering similar *concrete states* in the MAS model into a single *abstract state*. In order for the scheme to be practical, it must be easy to use. Moreover, it has to avoid the generation of the full concrete model, i.e., circumvent the complexity bottleneck. We will employ EASYABSTRACT — an open-source abstraction tool from Chapter 5 — that implements method proposed earlier in Chapter 4 to produce two abstract models a *may-abstraction* (that over-approximates the concrete states and transitions) and a *must-abstraction* (that under-approximates them). We remind that if an **ACTL*** formula is true in the *may*-abstraction, then it must be true in the concrete model, and if it is false in the *must*-abstraction, then it must be false in the concrete model.

8.4 Formal Models of SAI

In this section we describe our new formal models of SAI. The models are aimed at representing both the intended and adversarial behavior of PAIVs. The former is modeled through so-called "honest" AI agents. For the latter, we use two kinds of malicious AI agents: an "impersonator" and a "man-in-the-middle" attacker. Our new models have been strongly inspired by [KJS23], where SAI were specified using Asynchronous Multi-Agent Systems (AMAS) and verified using the STV model checker. In this work, we use MAS Graphs for the modelling part, and the UPPAAL model checker for verification. MAS Graphs allow for more flexibility than AMAS in the specification of the formal model. Moreover, UPPAAL better avoids the state-space explosion than STV. In consequence, we have been able to create and verify *richer* and more sophisticated models of SAI than [KJS23], e.g., by considering different topologies of sharing the machine learning models between agents. Moreover, temporal verification of MAS Graphs admits practical model reductions of [JK23a; JK23b], which we employ in this work to mitigate the complexity of the verification process.

A preliminary take on MAS Graph-based models and abstraction for SAI was reported Section 5.5.2, but that was mainly done to demonstrate the functionality of the abstraction tool, and considered only one particular variant of SAI models.

We begin with a high-level overview of the system and AI agent behaviour. Then, we provide several variants for the lower-level specification, which will further establish the scope for experiments in Section 8.5.

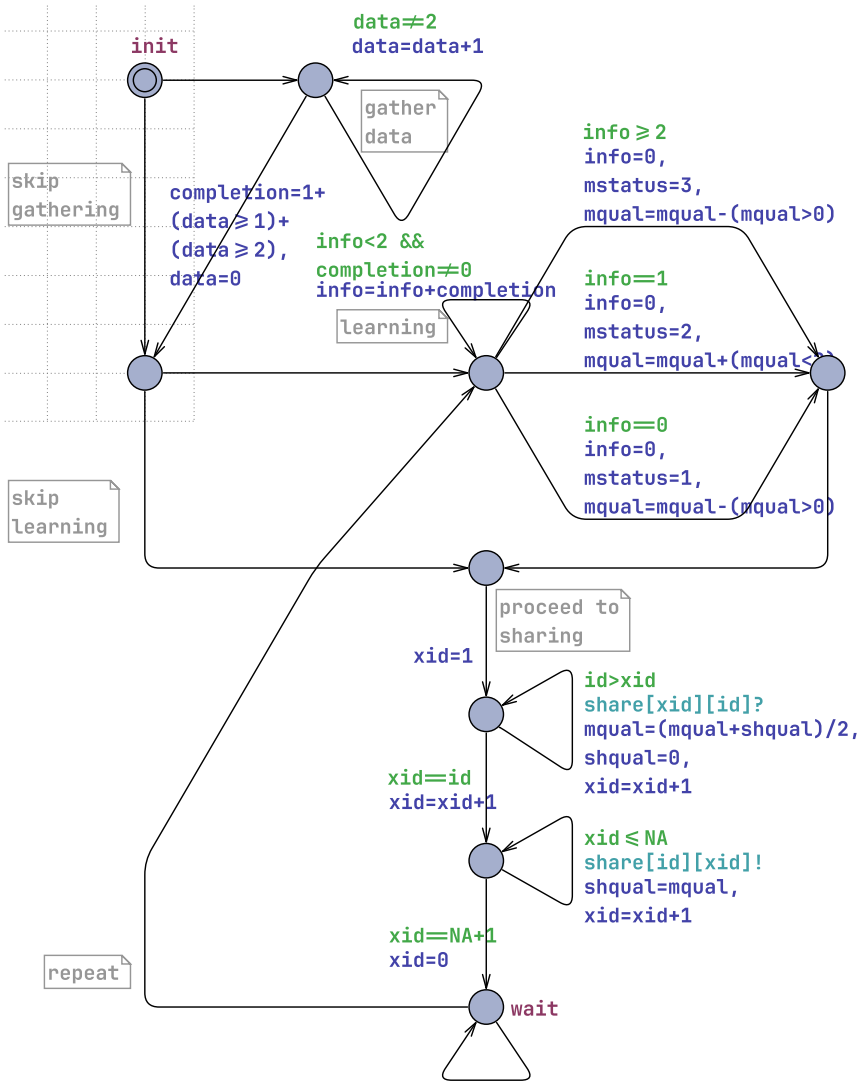


FIGURE 8.1: A template of AI agent in meta-configuration: reversed-cascade-network, sharing via average, no attacker.

8.4.1 AI Agents

The system is composed of a number of AI agents, each having a unique identifier. An example agent graph template for an AI agent is shown in Fig. 8.1.

The local model of an AI agent involves three subsequent phases: data gathering, learning and sharing. During the *data gathering* phase agent collects the data required for the learning. The amount of data is represented by a local variable `data`, which is incremented by taking the corresponding transition multiple times. When the gathering phase is finished, the data gets processed and categorized as either incomplete, complete or excessive. Next, in the *learning phase*, the agent proceeds

with training its machine learning component (ML-component in short), based on the previously acquired data. Depending on data completeness and the number of learning iterations, the quality of the ML-component is adjusted. Notably, the learning process does not affect the quality when no data has been acquired, and overtraining generally decreases the quality of the component.

It is also possible for an agent to completely skip data gathering and/or learning phases.

Afterwards, in the *sharing phase*, the agent shares its ML-component with other AI agents. Here we assume the case of asymmetric exchange, where the sender sends its ML-component and the receiver merges it with its own component. Which pairs of agents can communicate (and in which order) is specified by the network topology (see the examples in Section 8.4.2). Finally, the agent can return to the learning phase, or refrain from doing so.

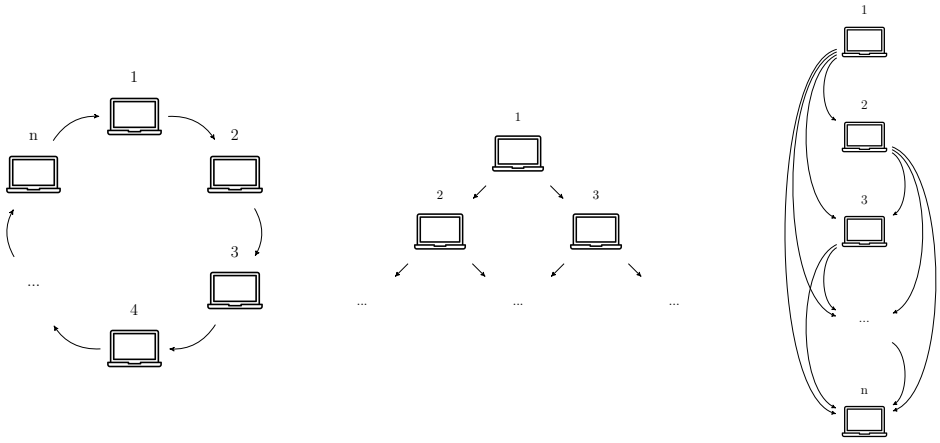


FIGURE 8.2: Informal illustration for possible message flow in (from left to right): ring, tree and reversed-cascade networks.

8.4.2 Scenarios

We consider several scenarios with different *meta-parameters*: the network topology (ring, tree, reversed-cascade), attacker type (none, man-in-the-middle, impersonator), and the operator for computing the outcome of sharing (minimum, average, maximum).

Topology. The network topology outlines the structure of communication between AI agents during the sharing phase. Selected variants are described below (see Fig. 8.2 for intuition):

- In the *ring-network*, each agent communicates with the pair of adjacent agents. Without loss of generality, we assume that agents with odd identifiers first transmit their model quality and then proceed to receive incoming models (and conversely for ones with odd identifiers).
- In the *tree-network*, messages are sent top-down, starting from a distinguished node, called the root. Each node has a single parent and can have up to n-children, where n denotes the arity. Here, we assume the case of complete binary trees, where all levels, except possibly the last, are filled.
- In the *reversed-cascade-network*, each node with identifier i receives messages from those with $\texttt{id} < i$ and then can start sending to those with $\texttt{id} > i$.

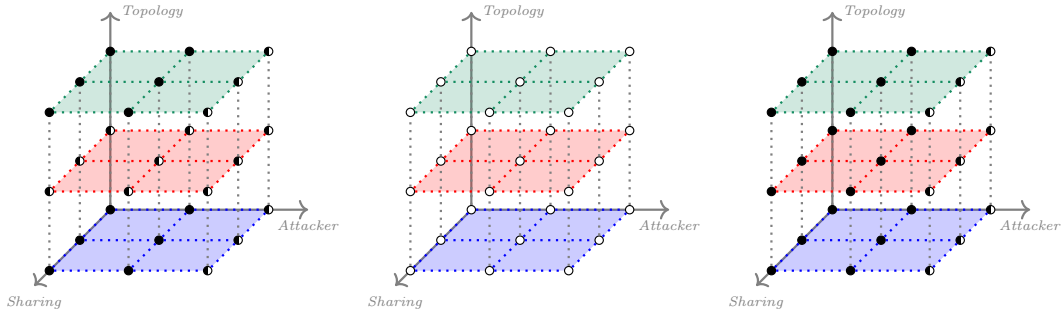


FIGURE 8.3: Verification results for model checking (from left to right): φ_1, φ_2 and φ_3. Nodes in the "cube" stand for possible meta-configurations of SAI models, (x, y, z) where coordinates map to specific element of (none, man-in-the-middle, impersonator) $\times$ (ring, tree, r-cascade) $\times$ (min, avg, max). Each node is denoted by: black-filled circle if given formula was satisfied on all attempted t-configurations, half-filled circle if it is satisfied for some, and empty circle if satisfied on none of them.

Sharing method. When an agent receives a machine learning component, it merges the component with that of its own. We specify the merging outcome by means of its effect on the resulting ML-component quality, taking either the maximum, the average, or the minimum of the original and the received model quality.

Attacker. In addition to a scenario with no attacker, where all agents are honest and follow the protocol as expected, we analyse those with an attacker. Here, we utilize two well-known types of adversary: man-in-the-middle and impersonator. Of course, this does not constitute a complete threat analysis, but already shows the way towards the verification of resistance against other, more sophisticated attacks.

- *Man-in-the-middle* attacker can intercept the communication and re-direct it but without any changes to the message content. As such an attacker may also abstain from interception, all executions that were present in the meta-configuration without an attacker will also be present here.
- *Impersonator* attacker acts in place of a selected AI agent. It only participates in the sharing phase(s) and exchanges messages as prescribed by its role. However, in contrast to an honest AI agent, an impersonator can forge an ML-component of any chosen quality prior to each transfer.

Altogether this gives 27 variants of MAS templates (see Fig. 8.3 for a graphical illustration), each being parameterized by the number of AI agents. A collection of instances from the same MAS template makes up a family of models. We use terms *meta-configuration* and *t-configuration* when referring to actual values of meta-parameters and template parameters respectively.

8.5 Experiments

We have performed a series of experiments with the aforementioned 27 families of SAI models. The experiments were conducted using UPPAAL v4.1.24 and EASYABSTRACT on a machine with Intel i7-8665U 2.11 GHz CPU, 16 GB RAM, running Ubuntu 22.04. The source code of the models (both concrete and abstract), as well as detailed results, can be found at https://tinyurl.com/sai-abstraction.

8.5.1 Requirement Specification

Deadlock-freeness. Deadlock occurs when neither system component can proceed. In other words, it is a global state with no outgoing transitions. Deadlock-freeness is achieved when the system is guaranteed to never reach a deadlock state. While some model checkers provide a special atomic proposition `deadlock` dedicated for deadlock states, the property can be also simulated within "vanilla" **ACTL***. For example, we can select some agent's location(s) and verify that it gets visited infinitely many times via the following formula:

$$\varphi_1 \equiv \mathsf{AG}(\mathsf{false} \Rightarrow \mathsf{AF} \bigvee_{i \in AI} \mathrm{wait}_i)$$

Note that location "wait" of the AI template in Fig. 8.1 has a self-loop, and thus it is guaranteed that there will be at least one outgoing transition. The above is a stronger requirement than AG¬deadlock; thus, when the former holds the latter must hold as well.

Eventually-win. Suppose that we want to verify if the SAI network is guaranteed to eventually reach a "winning" state where the average ML-component quality of the involved AI agents is greater than 0. This can be formalized by

$$\varphi_2 \equiv \mathsf{AF}(\mathrm{avg}(\mathrm{mqual}_i)_{i \in \mathrm{AI}} > 0)$$

In other words, we check if the system guarantees progress to a state better than the initial one. Clearly, other interpretations of a "win" can be interesting too. Similar queries can also facilitate the analysis of system modifications and design improvements. In order to force standard interpretation of AF formulas we introduce a benign modification to the models (just before verification) and appended each location with an invariant over the clock variable. Note, that doing this had no side effect on the state-space, and merely filtered non-maximal paths.

Flawless-wins. In a multi-agent system the goals of different agents (or their coalitions) are often conflicting. Therefore, a guaranteed achievement of all the goals (within every execution, no matter the chosen action) seldom happens. One of the common approaches is to reason about strategic abilities: whether there exists a strategy for the coalition that secures a win. Despite the limitation of UPPAAL that admits only verification of temporal properties, some results can still be obtained. For example, [JK23a] showed that, if the winning condition is free of modal operators, one can manually fix the candidate strategy, and then using UPPAAL check if it enforces a win. Here, instead of trying to guess the full strategy, we verify whether the achievement of certain sub-goals will ultimately guarantee winning. That is, we refine a previously introduced property and narrow down the scope of executions, where the winning state is expected to eventually occur. The formula

$$\mathsf{AG}(\bigwedge_{i \in \mathrm{AI}} \text{flawless-learner}_i \Rightarrow \varphi_2)$$

says that if all AI agents performed the learning phase perfectly then φ_2 is eventually guaranteed. For technical reasons, we also need to ignore runs, where agents self-loop in "wait" location and express "flawless-learning" by persistent evaluation of

#Ag	Concrete		Abstract **A1**			Abstract **A2**		
	#St	**t**	**#St**	**Reduct**	**t**	**#St**	**Reduct**	**t**
2	5832	0.1	81	72	0	53	110.03	0
3	363 013	2.3	625	580.8	0	327	1110.1	0
4	25 216 704	213	4851	5198.2	0	1995	12639.9	0
5	memout		37 790	–	0.2	12 014	–	0.1
6	memout		299 226	–	1.9	73 154	–	0.7
7	memout		2 374 295	–	23.7	443 593	–	5.9
8	memout		19 059 651	–	251.4	2 724 787	–	46.1
9	memout		memout			16 672 836	–	329.1
10	memout		memout			memout		

TABLE 8.1: Results of model checking φ_3 on meta-configuration with ring-network, sharing via average, no attacker. The column "#Ag" denotes the t-configuration (number of AI agents), "#St" number of states in a global model, "t" avg. verification time in seconds, and "Reduct" shows the level of reduction in the state space.

mstatus_i=2. These enhancements result in a formula

$$\begin{aligned}\varphi_3 \equiv \mathsf{AG}((\bigwedge_{i\in \mathrm{AI}'} \text{mstatus}_i{=}2) \Rightarrow \\ \mathsf{AF}((\bigwedge_{i\in \mathrm{AI}'} \text{mstatus}_i{=}2) \Rightarrow \\ (\text{avg}(\text{mqual}_i)_{i\in \mathrm{AI}'} > 0)))\end{aligned}$$

where $\mathrm{AI}' = \mathrm{AI}\backslash\{\text{impersonated}\}$ in the meta-configurations with impersonator, and $\mathrm{AI}' = \mathrm{AI}$ otherwise.

8.5.2 Dealing with State-Space Explosion

We have utilised the open-source experimental abstraction tool EASYABSTRACT[2], which automatically generates reduced formal models after applying the specified variable-based abstractions. A notable advantage of the tool is that it creates models that are portable. The output models are specified in the very same modular format as the input ones, and can be therefore opened, inspected and further used in UPPAAL; there is no side effect backwards dependence on a third-party tool afterwards.

We have employed the following abstractions:

A1. Removes variables `completion` and `mstatus` from AI agent templates;

A2. Removes variables `data`, `completion`, `mstatus`, `info` from AI agent templates;

We use the former for the verification of φ_1 and the latter for φ_3.[3] In both cases, over-approximating may-abstractions were conclusive.[4]

8.5.3 Results and Discussion

An aggregated view of the experimental results is shown in Fig. 8.3. We have been able to perform the verification of φ_1 (resp. φ_3) on concrete models with up to 4

[2] https://tinyurl.com/EasyAbstract4Uppaal

[3] Note that in order to verify φ_3, the abstractions **A1** and **A2** had to be slightly modified to consider the case of `mstatus==2` instead of actually removing the variable.

[4] It should also be noted that attempting to verify φ_1 on models from **A2** produces inconclusive results.

AI agents, and up to 8 (resp. 9) AI agents after applying abstraction **A1** (resp. **A2**). Notably, cases, when φ_1 and φ_2 were not satisfied, arise only for the t-configuration with one AI agent and only for meta-configurations that involved the Impersonator attacker or (in case of φ_1) the tree topology of the SAI network.

The verification of φ_2 resulted with "property not satisfied" in all the studied cases, and the model checker was able to quickly find and report a witnessing counter-example. Therefore, abstraction was not needed for this instance of verification.

Reasoning whether the same result would hold for a *whole* family of models (i.e., on every possible t-configuration) is generally much more challenging if feasible at all. To the best of our knowledge, there exists no universally applicable approach to achieve that. Nonetheless, a common conjecture[5] suggests that often it suffices to look for fairly small (violating) counter-examples. And whilst an absence of such counter-examples does not provide complete assurance, it does strengthen the confidence in the system being compliant with the requirements.

8.6 Conclusions

In this work, we have applied the formalism of MAS graphs, together with the branching-time specification of requirements, to formally model and verify Social Explainable AI (SAI). We constructed and studied 27 variants of scalable model families, further parameterized by the type and number of involved AI agents. This way, we showed how certain important properties could be specified using temporal logic and then verified in UPPAAL. Furthermore, we used a recently proposed user-friendly tool for practical abstraction EASYABSTRACT to demonstrate how to mitigate the state-space explosion. The reported results are very promising: in most cases we were able to double the number of agents that can be handled by the model checker before running out of memory.

In the future, we plan to conduct a more comprehensive analysis of the threats (e.g., consider other types of attack models) as well as capture more nuanced formulas. For example, one can use temporal-epistemic logic to express and verify *starvation-freeness*, which is a much stronger requirement than the basic notion of deadlock-freeness.

[5] For example, as in [ACK16].

Chapter 9

Conclusions

In the final chapter we provide a supplementary discussion, a summary of the thesis and propose possible directions for future work.

9.1 Discussion

The formalism proposed in Section 4.2 takes foundation from the concepts of modelling concurrent systems in [BK08] and [Cla+00a], and follows similar notational conventions. The syntax of structures for agent graphs, MAS graphs and combined MAS graphs, models and semantics of unwrapping share a lot of similarities with program graphs, interleaving of program graphs, transition systems and unfolding. We introduce these notions under a different name (instead of possibly referring to each as "a variant of . . .") for the sake of clarity and to further emphasize the distinct context of consideration (i.e., not solely that of software verification). In doing so, there was no intention to obscure or diminish the credit of the original work and its authors. Similarly, notions such as path, run, similarity relation, syntax and semantics for logics are already well-established in the literature and appear with marginal or no changes in their definition.

In general, the lower-approximation of local domain described in Algorithm 2 turned out to be of little use. From the correctness discussion in Section 4.3.2, it follows that obtained lower approximation for any given location is always either a singleton or an empty set. In practice, we propose to use may-abstraction instead and to inspect the abstract counter-example run (if any). An algorithm for finding concrete runs matching the abstract one is straightforward (see e.g., [Cla+00a]). Furthermore, we note that the construction of the must-abstraction remains correct and could be used with any valid lower-approximation of local domain.

The abstraction procedure from Chapter 4 was not applied for verification and experiments of Chapter 3 due to the system being fairly data-dependent, which significantly reduces the expected efficiency and rationale for the abstraction.

We also report no use of abstraction in Chapter 7. Firstly, the main focus of that work was to propose a more powerful modelling approach through the hierarchical and parameterized specification of the system. However, due to purely technical limitations of the current version of EASYABSTRACT tool from Chapter 5, it was incompatible with existing model specifications.[1] While it would certainly be interesting to present more sophisticated experimental results and evaluate the efficiency of the abstraction method from Chapter 4, we believe it does not undermine the soundness and appeal of the proposed approach. We are inclined to speculate that proposed hierarchical models should in fact be compatible with many other

[1] We plan to release an updated version that will provide the required functionalities in the future.

interesting reduction methods. A formal study of that could be a subject of future work.

9.2 Summary

First, we overview the voting processes, relevant properties and mechanisms, and cover some recent studies in Chapter 1. We excerpt the commonly used model representations, software tools and their supported logical formalism.

Next, the Chapter 2 aggregates the basic terminology and model checking primitives required for understanding the rest of the thesis.

Throughout the dissertation, various concepts and approaches to modelling were proposed and illustrated through motivational examples that were also intuitively explained. Those were largely based on existing frameworks and theoretical ideas proposed in the literature, adapted and lightened to serve the practical needs without overcomplicating the workflow.

Chapter 3 describes in detail the modelling and verification of the voting procedure on example of Prêt à Voter protocol. This work adopts the state-of-the-art model checker UPPAAL and highlights the advantages of having the GUI and flexible model specification language for the analysis of complex systems, such as voting. It also demonstrates the possible model extensions to accommodate examination of the Pfitzmann's attack, known to pose privacy risks with randomized mix-nets, as well as the simulation of knowledge operator under the **CTL** semantics.

In Chapter 4 we present a novel model reduction procedure that allows to compute a correct-by-design state-abstractions. The abstraction is parameterized by means of operations over the variables, each possibly narrowed to a certain fraction of states (specified by their underlying locations). No less importantly, when given arguments it can be performed in an automatized manner, and it provides certain guarantees that preserve the satisfiability of universal fragment of computation tree logic **ACTL***.

Chapter 5 implements the aforementioned abstraction method in a form of an open-source tool EASYABSTRACT. Notably, the produced abstract model specification remains compatible with UPPAAL and can be opened and edited in the usual way. The chapter outlines the architecture, provides intuitive scenarios for application, and practically evaluates its efficiency on example of postal voting and social AI models.

The Chapter 6 proposes a more extensive study of the Polish postal voting scenario. It shows how certain important socio-technical aspects of voting can be formally captured and verified. In particular, it demonstrates the steps necessary to mitigate the state space explosion by applying abstraction. Moreover, it improves on the visual clarity and manageability of model specification by employing the edge labelling convention. It suggests replacing the condition and update expressions, which are often technically complex, with the function calls on the corresponding event code, which could be represented as a string (i.e., an identifier name of a constant type variable). In consequence, not only was the overall readability of the model gets improved, but also the modularity of the specification. Nonetheless, it is also worth noting that with a larger number of events within the specification of a single agent (or its template), a description of its behaviour on the back-end (i.e., the body of aforementioned functions) would still descend into lengthy blocks of code.

The hierarchical approach discussed in Chapter 7 can be used to resolve that issue. It offers an even greater degree of modularity, encourages top-down design and natural partitioning of the specification on the structural level. This allows for switching between different behaviours of the agents involved without having to

create a duplicate model file and manually apply the relevant changes. Furthermore, it helps to distinguish between the knowledge of a fact and the possession of a physical object on the level of model specification. Object instances are associated with an owner agent, who can query or modify the object's attributes through defined interactions.

Lastly, Chapter 8 applies the MAS graph formalism and studies 27 variants of Social AI model families. It also demonstrates the applicability and efficiency of the proposed abstraction technique from Chapter 4 to protocols beyond voting schemes.

9.3 Future Work

In Chapters 3 to 8 we have already discussed some directions for future work. Those could be summarized as follows:

- Considering other verification tools that support a richer specification language for expressing system requirements. This includes tools that can verify strategic abilities and temporal epistemic properties. Although still in the early stages of development, STV appears to be a promising candidate for that.
- Developing a methodology for joint/complementary verification using different verification tools together would be of interest. One of the primary benefits of UPPAAL is its graphical user interface, which simplifies the process of writing system specifications and makes it more accessible. An automatic translation utility for the model specification from UPPAAL to different tools could also be useful.[2]
- Extending the abstraction methodology both *theoretically*, by adding the support for other kinds of operations, such as transformations over locations, and for other kinds of logics, in particular those capable of expressing epistemic and strategic properties, and *practically*, by adapting the implementation to other verification tools.
- It is evident that the complexity of verification increases rapidly with the greater number of agents involved. A possible track for future research can be to study the classes of voting scheme properties, for which saturation results can be obtained.

[2]The LTSMIN toolset utilises the Partitioned Next-State Interface (PINS) to achieve language independence and reports an implementation for UPPAAL specification language, which should enable the effective import of models of the latter. However, the conversion script relies on oppaal [Dal11], which is a separate tool that is no longer being developed. The latest version, released in 2013, contains several deprecated dependencies, offers limited support of UPPAAL syntax, and is poorly documented.

Bibliography

[ACK16] M. Arapinis, V. Cortier, and S. Kremer. "When Are Three Voters Enough for Privacy Properties?" In: *Proceedings of ESORICS*. Vol. 9879. Lecture Notes in Computer Science. Springer, 2016, pp. 241–260. DOI: 10.1007/978-3-319-45741-3_13.

[Adi08] Ben Adida. "Helios: web-based open-audit voting". In: *Proceedings of the 17th conference on Security symposium*. SS'08. San Jose, CA: USENIX Association, 2008, pp. 335–348.

[AF01] Martín Abadi and Cédric Fournet. "Mobile values, new names, and secure communication". In: *ACM Sigplan Notices* 36.3 (2001), pp. 104–115.

[AGJ04] Luca de Alfaro, Patrice Godefroid, and Radha Jagadeesan. "Three-Valued Abstractions of Games: Uncertainty, but with Precision". In: *Proceedings of Logic in Computer Science (LICS)*. IEEE Computer Society, 2004, pp. 170–179.

[AH99] R. Alur and T. A. Henzinger. "Reactive Modules". In: *Formal Methods in System Design* 15.1 (1999), pp. 7–48.

[AHK02] R. Alur, T. A. Henzinger, and O. Kupferman. "Alternating-Time Temporal Logic". In: *Journal of the ACM* 49 (2002), pp. 672–713. DOI: 10.1145/585265.585270.

[AHK97] R. Alur, T. A. Henzinger, and O. Kupferman. "Alternating-Time Temporal Logic". In: *Proceedings of the 38th Annual Symposium on Foundations of Computer Science (FOCS)*. IEEE Computer Society Press, 1997, pp. 100–109.

[Aki+20] Michael E. Akintunde, Elena Botoeva, Panagiotis Kouvaros, and Alessio Lomuscio. "Verifying Strategic Abilities of Neural-symbolic Multi-agent Systems". In: *Proceedings of the 17th International Conference on Principles of Knowledge Representation and Reasoning KR*. 2020, pp. 22–32. DOI: 10.24963/kr.2020/3.

[Alu] R Alur. "T.. A Henzinger, and O. Kupferman. Alternating-Time Temporal Logic". In: *38th Annual Symposium on Foundations of Computer Science (FOCS'97), Miami Beach, Florida, USA, October*, pp. 19–22.

[Alu+00] R. Alur, L. de Alfaro, T. A. Henzinger, S.C. Krishnan, F.Y.C. Mang, S. Qadeer, S.K. Rajamani, and S. Tasiran. *MOCHA: Modularity in Model Checking*. Tech. rep. University of Berkeley, 2000.

[Alu+98] R. Alur, T.A. Henzinger, O. Kupferman, and M.Y. Vardi. "Alternating refinement relations". In: *Proceedings of CONCUR*. Vol. 1466. Lecture Notes in Computer Science. 1998, pp. 163–178.

[Alu99] Rajeev Alur. "Timed automata". In: *Computer Aided Verification: 11th International Conference, CAV'99 Trento, Italy, July 6–10, 1999 Proceedings 11*. Springer. 1999, pp. 8–22.

[AM07] Michael R Clarkson Stephen Chong Andrew and C Myers. "Civitas: A secure remote voting system". In: (2007).

[And20] Marcin Skubiszewski Andrzej Rzepliński. *Oświadczenia w sprawie nieprawidłowości dotyczących wyborów [translated: Statement on election irregularities]*. 2020-08-05. URL: https://ow.org.pl/2020/08/05/oswiadczenia-w-sprawie-nieprawidlowosci-dotyczacych-wyborow/ (visited on 2022-04-27).

[Ara+13] Myrto Arapinis, Véronique Cortier, Steve Kremer, and Mark Ryan. "Practical everlasting privacy". In: *International Conference on Principles of Security and Trust*. Springer. 2013, pp. 21–40.

[Arm+05] Alessandro Armando, David Basin, Yohan Boichut, Yannick Chevalier, Luca Compagna, Jorge Cuéllar, P Hankes Drielsma, Pierre-Cyrille Héam, Olga Kouchnarenko, Jacopo Mantovani, et al. "The AVISPA tool for the automated validation of internet security protocols and applications". In: *Computer Aided Verification: 17th International Conference, CAV 2005, Edinburgh, Scotland, UK, July 6-10, 2005. Proceedings 17*. Springer. 2005, pp. 281–285. DOI: 10.1007/11513988_27.

[BAF08] Bruno Blanchet, Martín Abadi, and Cédric Fournet. "Automated verification of selected equivalences for security protocols". In: *The Journal of Logic and Algebraic Programming* 75.1 (2008), pp. 3–51.

[Bas+17] David A. Basin, Hans Gersbach, Akaki Mamageishvili, Lara Schmid, and Oriol Tejada. "Election Security and Economics: It's All About Eve". In: *Proceedings of E-Vote-ID*. 2017, pp. 1–20. DOI: 10.1007/978-3-319-68687-5_1.

[BB06] Bernhard Beckert and Gerd Beuster. "A method for formalizing, analyzing, and verifying secure user interfaces". In: *International Conference on Formal Engineering Methods*. Springer. 2006, pp. 55–73.

[BCG04] Elwyn R Berlekamp, John H Conway, and Richard K Guy. *Winning ways for your mathematical plays, volume 4*. AK Peters/CRC Press, 2004.

[BCL15] Giampaolo Bella, Paul Curzon, and Gabriele Lenzini. "Service security and privacy as a socio-technical problem". In: *J. Comput. Secur.* 23.5 (2015), pp. 563–585. DOI: 10.3233/JCS-150536.

[BDJ10] N. Bulling, J. Dix, and W. Jamroga. "Model Checking Logics of Strategic Ability: Complexity". In: *Specification and Verification of Multi-Agent Systems*. Ed. by M. Dastani, K. Hindriks, and J.-J. Meyer. Springer, 2010, pp. 125–159.

[BDL04] G. Behrmann, A. David, and K.G. Larsen. "A Tutorial on UPPAAL". In: *Formal Methods for the Design of Real-Time Systems: SFM-RT*. LNCS 3185. Springer, 2004, pp. 200–236.

[BDS17] A. Bruni, E. Drewsen, and C. Schürmann. "Towards a Mechanized Proof of Selene Receipt-Freeness and Vote-Privacy". In: *Proceedings of E-Vote-ID*. Vol. 10615. Lecture Notes in Computer Science. Springer, 2017, pp. 110–126. DOI: 10.1007/978-3-319-68687-5_7.

[Bec+16] B. Beckert, M. Kirsten, V. Klebanov, and C. Schürmann. "Automatic Margin Computation for Risk-Limiting Audits". In: *Proceedings of E-Vote-ID*. Vol. 10141. Lecture Notes in Computer Science. Springer, 2016, pp. 18–35. DOI: 10.1007/978-3-319-52240-1_2.

[Bel+13] Susan Bell, Josh Benaloh, Michael D Byrne, Dana DeBeauvoir, Bryce Eakin, Philip Kortum, Neal McBurnett, Olivier Pereira, Philip B Stark, Dan S Wallach, et al. "{STAR-Vote}: A secure, transparent, auditable, and reliable voting system". In: *2013 Electronic Voting Technology Workshop/Workshop on Trustworthy Elections (EVT/WOTE 13)*. 2013.

[Bel+14] Giampaolo Bella, Paul Curzon, Rosario Giustolisi, and Gabriele Lenzini. "A Socio-technical Methodology for the Security and Privacy Analysis of Services". In: *COMPSAC Workshops*. IEEE Computer Society, 2014, pp. 401–406. DOI: 10.1109/COMPSACW.2014.69.

[Bel+21] Francesco Belardinelli, Rodica Condurache, Catalin Dima, Wojciech Jamroga, and Michal Knapik. "Bisimulations for verifying strategic abilities with an application to the ThreeBallot voting protocol". In: *Information and Computation* 276 (2021), p. 104552. DOI: 10.1016/j.ic.2020.104552.

[Ben+15] Josh Benaloh, Ronald Rivest, Peter YA Ryan, Philip Stark, Vanessa Teague, and Poorvi Vora. "End-to-end verifiability". In: *arXiv preprint arXiv:1504.03778* (2015).

[Ben06] Josh Benaloh. "Simple Verifiable Elections". In: *USENIX Electronic Voting Technology Workshop*. 2006.

[Ben07] Josh Benaloh. "Ballot Casting Assurance via Voter-Initiated Poll Station Auditing". In: *USENIX/ACCURATE Electronic Voting Technology Workshop*. 2007.

[Ber+04] Y. Bertot, P. Casteran, G. Huet, and C. Paulin-Mohring. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Springer, 2004. DOI: 10.1007/978-3-662-07964-5.

[Ber+15] David Bernhard, Véronique Cortier, David Galindo, Olivier Pereira, and Bogdan Warinschi. "SoK: A comprehensive analysis of game-based ballot privacy definitions". In: *2015 IEEE Symposium on Security and Privacy*. IEEE. 2015, pp. 499–516.

[BGJ15] N. Bulling, V. Goranko, and W. Jamroga. "Logics for Reasoning About Strategic Abilities in Multi-Player Games". In: *Models of Strategic Reasoning. Logics, Games, and Communities*. Vol. 8972. Lecture Notes in Computer Science. Springer, 2015, pp. 93–136. DOI: 10.1007/978-3-662-48540-8.

[BGS13] B. Beckert, R. Goré, and C. Schürmann. "Analysing Vote Counting Algorithms via Logic - And Its Application to the CADE Election Scheme". In: *Proceedings of CADE*. Vol. 7898. Lecture Notes in Computer Science. Springer, 2013, pp. 135–144. DOI: 10.1007/978-3-642-38574-2_9.

[BGS22] Giampaolo Bella, Rosario Giustolisi, and Carsten Schürmann. "Modelling human threats in security ceremonies". In: *Journal of Computer Security* Preprint (2022), pp. 1–23.

[BK06] T. Ball and O. Kupferman. "An Abstraction-Refinement Framework for Multi-Agent Systems". In: *Proceedings of Logic in Computer Science (LICS)*. IEEE, 2006, pp. 379–388. DOI: 10.1109/LICS.2006.10.

[BK08] Christel Baier and Joost-Pieter Katoen. *Principles of model checking*. MIT press, 2008.

[BKL16] I. Boureanu, P. Kouvaros, and A. Lomuscio. "Verifying Security Properties in Unbounded Multiagent Systems". In: *Proceedings of International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS)*. 2016, pp. 1209–1217.

[BKL17] Francesco Belardinelli, Panagiotis Kouvaros, and Alessio Lomuscio. "Parameterised Verification of Data-aware Multi-Agent Systems". In: *Proceedings of IJCAI*. ijcai.org, 2017, pp. 98–104. DOI: 10.24963/ijcai.2017/15.

[BL17] Francesco Belardinelli and Alessio Lomuscio. "Agent-based Abstractions for Verifying Alternating-time Temporal Logic with Imperfect Information". In: *Proceedings of AAMAS*. ACM, 2017, pp. 1259–1267.

[Bla+16] Bruno Blanchet et al. "Modeling and verifying security protocols with the applied pi calculus and ProVerif". In: *Foundations and Trends® in Privacy and Security* 1.1-2 (2016), pp. 1–135.

[BLM19] Francesco Belardinelli, Alessio Lomuscio, and Vadim Malvone. "An Abstraction-Based Method for Verifying Strategic Properties in Multi-Agent Systems with Imperfect Information". In: *Proceedings of AAAI*. 2019, pp. 6030–6037.

[BLP11] Francesco Belardinelli, Alessio Lomuscio, and Fabio Patrizi. "Verification of Deployed Artifact Systems via Data Abstraction". In: *Proceedings of ICSOC*. Vol. 7084. Lecture Notes in Computer Science. Springer, 2011, pp. 142–156. DOI: 10.1007/978-3-642-25535-9_10.

[BM07] Ahto Buldas and Triinu Mägi. "Practical Security Analysis of E-Voting Systems". In: *Proceedings of IWSEC*. Vol. 4752. Lecture Notes in Computer Science. Springer, 2007, pp. 320–335.

[BRS16] David A. Basin, Sasa Radomirovic, and Lara Schmid. "Modeling Human Errors in Security Protocols". In: *Computer Security Foundations Symposium, CSF*. IEEE Computer Society, 2016, pp. 325–340. DOI: 10.1109/CSF.2016.30.

[BRT13] Josh Benaloh, Peter Y.A. Ryan, and Vanessa Teague. "Verifiable postal voting". In: *Cambridge International Workshop on Security Protocols*. Springer. 2013, pp. 54–65.

[Bru+21] Alessandro Bruni, Marco Carbone, Rosario Giustolisi, Sebastian Mödersheim, and Carsten Schürmann. "Security Protocols as Choreographies". In: *Protocols, Strands, and Logic - Essays Dedicated to Joshua Guttman on the Occasion of his 66.66th Birthday*. Vol. 13066. Lecture Notes in Computer Science. Springer, 2021, pp. 98–111. DOI: 10.1007/978-3-030-91631-2_5.

[BT94] Josh Benaloh and Dwight Tuinstra. "Receipt-free secret-ballot elections". In: *Proceedings of the twenty-sixth annual ACM symposium on Theory of Computing*. ACM. 1994, pp. 544–553.

[Bur+12] C. Burton, C. Culnane, J. Heather, T. Peacock, P.Y.A. Ryan, S. Schneider, V. Teague, R. Wen, Z. Xia, and S. Srinivasan. "Using Prêt à Voter in Victoria State Elections". In: *Proceedings of EVT/WOTE*. USENIX, 2012.

[BY03] Johan Bengtsson and Wang Yi. "Timed automata: Semantics, algorithms and tools". In: *Advanced Course on Petri Nets*. Springer. 2003, pp. 87–124.

[Car+12] Marcelo Carlomagno Carlos, Jean Everson Martina, Geraint Price, and Ricardo Felipe Custódio. "A Proposed Framework for Analysing Security Ceremonies". In: *SECRYPT*. SciTePress, 2012, pp. 440–445.

[CB13] V. Cheval and B. Blanchet. "Proving More Observational Equivalences with ProVerif". In: *Proceedings of POST*. Vol. 7796. Lecture Notes in Computer Science. Springer, 2013, pp. 226–246. DOI: 10.1007/978-3-642-36830-1_12.

[CC77] P. Cousot and R. Cousot. "Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints". In: *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages*. 1977, pp. 238–252. DOI: 10.1145/512950.512973.

[CES86] Edmund M Clarke, E Allen Emerson, and A Prasad Sistla. "Automatic verification of finite-state concurrent systems using temporal logic specifications". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 8.2 (1986), pp. 244–263.

[CFL19] Véronique Cortier, Alicia Filipiak, and Joseph Lallemand. "BeleniosVS: Secrecy and verifiability against a corrupted voting device". In: *2019 IEEE 32nd Computer Security Foundations Symposium (CSF)*. IEEE. 2019, pp. 367–36714.

[CGG19] Véronique Cortier, Pierrick Gaudry, and Stéphane Glondu. "Belenios: a simple private and verifiable electronic voting system". In: *Foundations of Security, Protocols, and Equational Reasoning: Essays Dedicated to Catherine A. Meadows* (2019), pp. 214–238.

[CGL94] E.M. Clarke, O. Grumberg, and D.E. Long. "Model Checking and Abstraction". In: *ACM Transactions on Programming Languages and Systems* 16.5 (1994), pp. 1512–1542.

[CGT18] Véronique Cortier, David Galindo, and Mathieu Turuani. "A formal analysis of the Neuchâtel e-voting protocol". In: *2018 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE. 2018, pp. 430–442.

[CGY22] Véronique Cortier, Pierrick Gaudry, and Quentin Yang. "Is the JCJ voting system really coercion-resistant?" In: *Cryptology ePrint Archive* (2022).

[Cha+09] D. Chaum, R.T. Carback, J. Clark, A. Essex, S. Popoveniuc, R.L. Rivest, P.Y.A. Ryan, E. Shen, A.T. Sherman, and P.L. Vora. "Scantegrity II: end-to-end verifiability by voters of optical scan elections through confirmation codes". In: *Trans. Info. For. Sec.* 4.4 (2009), pp. 611–627. ISSN: 1556-6013. DOI: 10.1109/TIFS.2009.2034919.

[Che+13] T. Chen, V. Forejt, M. Kwiatkowska, D. Parker, and A. Simaitis. "PRISM-games: A Model Checker for Stochastic Multi-Player Games". In: *Proceedings of Tools and Algorithms for Construction and Analysis of Systems (TACAS)*. Vol. 7795. Lecture Notes in Computer Science. Springer, 2013, pp. 185–191.

[Che+22] Vincent Cheval, Charlie Jacomme, Steve Kremer, and Robert Künnemann. "{SAPIC+}: protocol verifiers of the world, unite!" In: *31st USENIX Security Symposium (USENIX Security 22)*. 2022, pp. 3935–3952.

[Cim+02] A. Cimatti, E.M. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, M. Sebastiani, and A Tacchella. "NuSMV2: An Open-Source Tool for Symbolic Model Checking". In: *Proceedings of Computer Aided Verification (CAV)*. Vol. 2404. Lecture Notes in Computer Science. 2002, pp. 359–364.

[CKL04] E.M. Clarke, D. Kroening, and F. Lerda. "A Tool for Checking ANSI-C Programs". In: *Proceedings of TACAS*. Vol. 2988. Lecture Notes in Computer Science. Springer, 2004, pp. 168–176. DOI: 10.1007/978-3-540-24730-2_15.

[CKM01] Søren Christensen, Lars Michael Kristensen, and Thomas Mailund. "A sweep-line method for state space exploration". In: *Tools and Algorithms for the Construction and Analysis of Systems: 7th International Conference, TACAS 2001 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2001 Genova, Italy, April 2–6, 2001 Proceedings 7*. Springer. 2001, pp. 450–464.

[CKO22] Pierluigi Contucci, Janos Kertesz, and Godwin Osabutey. "Human-AI ecosystem with abrupt changes as a function of the composition". In: *PLOS ONE* 17.5 (2022-05), pp. 1–12. DOI: 10.1371/journal.pone.0267310.

[Cla+00a] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. "Counterexample-guided abstraction refinement". In: *International Conference on Computer Aided Verification*. Springer. 2000, pp. 154–169.

[Cla+00b] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. "Counterexample-Guided Abstraction Refinement". In: *Proceedings of CAV*. Vol. 1855. Lecture Notes in Computer Science. Springer, 2000, pp. 154–169. DOI: 10.1007/10722167_15.

[Cla+03] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. "Counterexample-guided abstraction refinement for symbolic model checking". In: *J. ACM* 50.5 (2003), pp. 752–794. DOI: 10.1145/876638.876643.

[Cla+18] E.M. Clarke, T.A. Henzinger, H. Veith, and R. Bloem, eds. *Handbook of Model Checking*. Springer, 2018. ISBN: 978-3-319-10574-1. DOI: 10.1007/978-3-319-10575-8.

[CM12] C. Cremers and S. Mauw. *Operational Semantics and Verification of Security Protocols*. Information Security and Cryptography. Springer, 2012. ISBN: 978-3-540-78636-8. DOI: 10.1007/978-3-540-78636-8.

[Coh+09] Mika Cohen, Mads Dam, Alessio Lomuscio, and Francesco Russo. "Abstraction in model checking multi-agent systems". In: *Proceedings of AAMAS*. IFAAMAS, 2009, pp. 945–952.

[Com24] Computing Research and Education Association of Australasia, CORE Incorporated. *CORE Rankings Portal*. https://www.core.edu.au/conference-portal. 2020–2024.

[Cor+09] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to algorithms*. MIT press, 2009.

[Cor+16] V. Cortier, D. Galindo, R. Küsters, J. Müller, and T. Truderung. "SoK: Verifiability Notions for E-Voting Protocols". In: *IEEE Symposium on Security and Privacy*. 2016, pp. 779–798. DOI: 10.1109/SP.2016.52.

[CP18] Marco Conti and Andrea Passarella. "The Internet of People: A human and data-centric paradigm for the Next Generation Internet". In: *Comput. Commun.* 131 (2018), pp. 51–65. DOI: 10.1016/j.comcom.2018.07.034.

[CRS05] David Chaum, Peter Y. A. Ryan, and Steve A. Schneider. "A Practical Voter-Verifiable Election Scheme". In: *Proceedings of ESORICS*. 2005, pp. 118–139. DOI: 10.1007/11555827_8.

[CT16] Chris Culnane and Vanessa Teague. "Strategies for Voter-Initiated Election Audits". In: *Decision and Game Theory for Security: Proceedings of GameSec*. Vol. 9996. Lecture Notes in Computer Science. Springer, 2016, pp. 235–247. DOI: 10.1007/978-3-319-47413-7_14.

[Cul+15] C. Culnane, P.Y.A. Ryan, S.A. Schneider, and V. Teague. "vVote: A Verifiable Voting System". In: *ACM Trans. Inf. Syst. Secur.* 18.1 (2015), 3:1–3:30. DOI: 10.1145/2746338.

[Cze+19] W. Czerwiński, S. Lasota, R. Lazić, J. Leroux, and F. Mazowiecki. "The Reachability Problem for Petri Nets is Not Elementary". In: *Proceedings of the 51st Annual ACM SIGACT Symposium on Theory of Computing STOC*. Association for Computing Machinery, 2019, pp. 24–33. DOI: 10.1145/3313276.3316369.

[Dal11] Andreas Engelbredt Dalsgaard. "opaal: A lattice model checker". In: *NASA Formal Methods: Third International Symposium, NFM 2011, Pasadena, CA, USA, April 18-20, 2011. Proceedings 3*. Springer. 2011, pp. 487–493.

[de +84] J.W. de Bakker, J.A. Bergstra, Jan Willem Klop, and J.-J. Ch. Meyer. "Linear Time and Branching Time Semantics for Recursion with Merge". In: *Theor. Comput. Sci.* 34 (1984), pp. 135–156. DOI: 10.1016/0304-3975(84)90114-2.

[Dem+03] P. Dembiński, A. Janowska, P. Janowski, W. Penczek, A. Półrola, M. Szreter, B. Woźna, and A. Zbrzezny. "Verics: A Tool for Verifying Timed Automata and Estelle Specifications". In: *Proceedings of the of the 9th Int. Conf. on Tools and Algorithms for Construction and Analysis of Systems (TACAS'03)*. Vol. 2619. Lecture Notes in Computer Science. Springer, 2003, pp. 278–283.

[DG18] Dennis Dams and Orna Grumberg. "Abstraction and Abstraction Refinement". In: *Handbook of Model Checking*. Springer, 2018, pp. 385–419. DOI: 10.1007/978-3-319-10575-8_13.

[DGG97] Dennis Dams, Rob Gerth, and Orna Grumberg. "Abstract Interpretation of Reactive Systems". In: *ACM Trans. Program. Lang. Syst.* 19.2 (1997), pp. 253–291. DOI: 10.1145/244795.244800. URL: https://doi.org/10.1145/244795.244800.

[DHM10] M. Dastani, K. Hindriks, and J.-J. Meyer, eds. *Specification and Verification of Multi-Agent Systems*. Springer, 2010.

[DK20] Piotr Dowbor and Yan Kim. "Computational Classification of Tubular Algebras". In: *Fundamenta Informaticae* 177.1 (2020), pp. 39–67.

[DKR06] S. Delaune, S. Kremer, and M. Ryan. "Coercion-resistance and receipt-freeness in electronic voting". In: *Computer Security Foundations Workshop, 2006. 19th IEEE*. IEEE. 2006, 12–pp.

[DLL12] Jannik Dreier, Pascal Lafourcade, and Yassine Lakhnech. "A formal taxonomy of privacy in voting protocols". In: *2012 IEEE International Conference on Communications (ICC)*. IEEE. 2012, pp. 6710–6715.

[Dra+20] Georgios Drainakis, Konstantinos V. Katsaros, Panagiotis Pantazopoulos, Vasilis Sourlas, and Angelos Amditis. "Federated vs. Centralized Machine Learning under Privacy-elastic Users: A Comparative Analysis". In: *Proceedings of NCA*. IEEE, 2020, pp. 1–8. DOI: 10.1109/NCA51143.2020.9306745.

[DY83] Danny Dolev and Andrew Chi-Chih Yao. "On the security of public key protocols". In: *IEEE Trans. Inf. Theory* 29.2 (1983), pp. 198–207. DOI: 10.1109/TIT.1983.1056650.

[ED08] C. Enea and C. Dima. "Abstractions of multi-agent systems". In: *International Transactions on Systems Science and Applications* 3.4 (2008), pp. 329–337.

[EH86] E Allen Emerson and Joseph Y Halpern. ""Sometimes" and "not never" revisited: on branching versus linear time temporal logic". In: *Journal of the ACM (JACM)* 33.1 (1986), pp. 151–178.

[Eme90] E.A. Emerson. "Temporal and Modal Logic". In: *Handbook of Theoretical Computer Science*. Ed. by J. van Leeuwen. Vol. B. Elsevier, 1990, pp. 995–1072.

[Fag+95] R. Fagin, J. Y. Halpern, Y. Moses, and M. Y. Vardi. *Reasoning about Knowledge*. MIT Press, 1995.

[Fak20] Fakt.pl editors. *Dostali karty do głosowania bez pieczęci. Czy głosy będą nieważne?* 2020-06-28. URL: https://www.fakt.pl/wydarzenia/polityka/dostali-karty-do-glosowania-bez-pieczeci-czy-glosy-beda-niewazne/6cwhzg4 (visited on 2022-05-14).

[FOO93] Atsushi Fujioka, Tatsuaki Okamoto, and Kazuo Ohta. "A practical secret voting scheme for large scale elections". In: *Advances in Cryptology—AUSCRYPT'92: Workshop on the Theory and Application of Cryptographic Techniques Gold Coast, Queensland, Australia, December 13–16, 1992 Proceedings 3*. Springer. 1993, pp. 244–251.

[FPC22] Andrew Fuchs, Andrea Passarella, and Marco Conti. "Modeling Human Behavior Part I - Learning and Belief Approaches". In: *CoRR* abs/2205.06485 (2022). DOI: 10.48550/arXiv.2205.06485. arXiv: 2205.06485.

[Ger+99] R. Gerth, R. Kuiper, D. Peled, and W. Penczek. "A Partial Order Approach to Branching Time Logic Model Checking". In: *Proceedings of ISTCS*. IEEE, 1999, pp. 130–139.

[Gha+18] M.K. Ghale, R. Goré, D. Pattinson, and M. Tiwari. "Modular Formalisation and Verification of STV Algorithms". In: *Proceedings of E-Vote-ID*. Vol. 11143. Lecture Notes in Computer Science. Springer, 2018, pp. 51–66. DOI: 10.1007/978-3-030-00419-4_4.

[GHJ01] P. Godefroid, M. Huth, and R. Jagadeesan. "Abstraction-Based Model Checking Using Modal Transition Systems". In: *Proceedings of CONCUR*. Vol. 2154. Lecture Notes in Computer Science. 2001, pp. 426–440.

[GJ02] P. Godefroid and R. Jagadeesan. "Automatic Abstraction Using Generalized Model Checking". In: *Proceedings of Computer Aided Verification (CAV)*. Vol. 2404. Lecture Notes in Computer Science. Springer, 2002, pp. 137–150. DOI: 10.1007/3-540-45657-0_11.

[GJ04] V. Goranko and W. Jamroga. "Comparing Semantics of Logics for Multi-agent Systems". In: *Synthese* 139.2 (2004), pp. 241–280.

[GL94] Orna Grumberg and David E Long. "Model checking and modular verification". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 16.3 (1994), pp. 843–871.

[Głó] Główny Urząd Statystyczny [translated: Statistics Poland]. *Słownik pojęć [translated: Glossary]*. URL: https://stat.gov.pl/metainformacje/slownik-pojec/ (visited on 2023-03-31).

[GMP18] Ian J. Goodfellow, Patrick D. McDaniel, and Nicolas Papernot. "Making machine learning robust against adversarial inputs". In: *Commun. ACM* 61.7 (2018), pp. 56–66. DOI: 10.1145/3134599.

[God+10] Patrice Godefroid, Aditya V. Nori, Sriram K. Rajamani, and SaiDeep Tetali. "Compositional may-must program analysis: unleashing the power of alternation". In: *Proceedings of POPL*. ACM, 2010, pp. 43–56. DOI: 10.1145/1706299.1706307.

[God14] Patrice Godefroid. "May/Must Abstraction-Based Software Model Checking for Sound Verification and Falsification". In: *Software Systems Safety*. Ed. by Orna Grumberg, Helmut Seidl, and Maximilian Irlbeck. Vol. 36. NATO Science for Peace and Security Series, D: Information and Communication Security. IOS Press, 2014, pp. 1–16. DOI: 10.3233/978-1-61499-385-8-1. URL: https://doi.org/10.3233/978-1-61499-385-8-1.

[God96] Patrice Godefroid. *Partial-order methods for the verification of concurrent systems: an approach to the state-explosion problem*. Springer, 1996.

[Gol11] Dieter Gollmann. *Computer Security (3. ed.)* Wiley, 2011. ISBN: 978-0-470-74115-3.

[GS92] S.M. German and A.P. Sistla. "Reasoning about Systems with Many Processes". In: *Journal of the ACM* 39.3 (1992), pp. 675–735. DOI: 10.1145/146637.146681.

[GWC06] Arie Gurfinkel, Ou Wei, and Marsha Chechik. "Yasm: A Software Model-Checker for Verification and Refutation". In: *Proceedings of CAV*. Vol. 4144. Lecture Notes in Computer Science. Springer, 2006, pp. 170–174. DOI: 10.1007/11817963_18.

[Hai+23] Thomas Haines, Johannes Mueller, Rafieh Mosaheb, and Ivan Pryvalov. "SoK: Secure e-voting with everlasting privacy". In: *Proceedings on Privacy Enhancing Technologies (PoPETs)* (2023).

[HDJ19] István Hegedüs, Gábor Danner, and Márk Jelasity. "Gossip Learning as a Decentralized Alternative to Federated Learning". In: *Proceedings of IFIP DAIS*. Vol. 11534. Lecture Notes in Computer Science. Springer, 2019, pp. 74–90. DOI: 10.1007/978-3-030-22496-7_5.

[HDJ21] István Hegedüs, Gábor Danner, and Márk Jelasity. "Decentralized learning works: An empirical comparison of gossip learning and federated learning". In: *J. Parallel Distributed Comput.* 148 (2021), pp. 109–124. DOI: 10.1016/j.jpdc.2020.10.006.

[Hea13] Douglas Heaven. "Social AI likes to gossip". In: *New Scientist* 218.2923 (2013), p. 20. ISSN: 0262-4079. DOI: https://doi.org/10.1016/S0262-4079(13)61601-2.

[Hen+94] Thomas A Henzinger, Xavier Nicollin, Joseph Sifakis, and Sergio Yovine. "Symbolic model checking for real-time systems". In: *Information and computation* 111.2 (1994), pp. 193–244.

[HGS21] Thomas Haines, Rajeev Goré, and Bhavesh Sharma. "Did you mix me? Formally Verifying Verifiable Mix Nets in Electronic Voting". In: *42nd IEEE Symposium on Security and Privacy, SP 2021, San Francisco, CA, USA, 24-27 May 2021*. IEEE, 2021, pp. 1748–1765. DOI: 10.1109/SP40001.2021.00033.

[HGT19] T. Haines, R. Goré, and M. Tiwari. "Verified Verifiers for Verifying Elections". In: *Proceedings of CCS*. ACM, 2019, pp. 685–702. DOI: 10.1145/3319535.3354247.

[Hoa13] Thai Son Hoang. "An introduction to the Event-B modelling method". In: *Industrial Deployment of System Engineering Methods* (2013), pp. 211–236.

[Hol21] Matthew Holroyd. "Dutch election: Rule change to accept wrongly sealed mail-in ballots". In: *Euronews* (2021-03-17). URL: https://www.euronews.com/2021/03/17/dutch-election-rule-change-to-accept-wrongly-sealed-mail-in-ballots (visited on 2022-04-17).

[HR16] F. Hao and P.Y.A. Ryan. *Real-World Electronic Voting: Design, Analysis and Deployment*. Auerbach Publications, 2016. ISBN: 1498714692.

[Hv14] X. Huang and R. van der Meyden. "Symbolic Model Checking Epistemic Strategy Logic". In: *Proceedings of AAAI Conference on Artificial Intelligence*. 2014, pp. 1426–1432.

[Jam+20a] W. Jamroga, W. Penczek, T. Sidoruk, P. Dembiński, and A. Mazurkiewicz. "Towards Partial Order Reductions for Strategic Ability". In: *Journal of Artificial Intelligence Research* 68 (2020), pp. 817–850. DOI: 10.1613/jair.1.11936.

[Jam+20b] Wojciech Jamroga, Yan Kim, Damian Kurpiewski, and Peter Y. A. Ryan. "Towards Model Checking of Voting Protocols in Uppaal". In: *Proceedings of E-Vote-ID*. Vol. 12455. Lecture Notes in Computer Science. Springer, 2020, pp. 129–146. DOI: 10.1007/978-3-030-60347-2_9.

[Jam+21] Wojciech Jamroga, David Mestel, Peter B. Roenne, Peter Y. A. Ryan, and Marjan Skrobot. "A Survey of Requirements for COVID-19 Mitigation Strategies". In: *Bulletin of The Polish Academy of Sciences: Technical Science* 69.4 (2021), e137724. DOI: 10.24425/bpasts.2021.137724.

[Jam+24] Wojciech Jamroga, Peter B. Roenne, Peter Y. A. Ryan, and Yan Kim. "You Shall not Abstain! A Formal Study of Forced Participation". In: *Proceeding of the 9th Workshop on Advances in Secure Electronic Voting, Voting 2024*. To appear. 2024.

[Jam08] W. Jamroga. "Knowledge and Strategic Ability for Model Checking: A Refined Approach". In: *Proceedings of MATES'08*. Vol. 5244. Lecture Notes in Computer Science. 2008, pp. 99–110.

[Jam15] W. Jamroga. *Logical Methods for Specification and Verification of Multi-Agent Systems*. ICS PAS Publishing House, 2015. ISBN: 978-83-63159-25-2.

[Jam23] Wojciech Jamroga. "Pretty Good Strategies for Benaloh Challenge". In: *International Joint Conference on Electronic Voting*. Springer. 2023, pp. 106–122.

[JCJ05] A. Juels, D. Catalano, and M. Jakobsson. "Coercion-resistant electronic elections". In: *Proceedings of the 2005 ACM workshop on Privacy in the electronic society*. ACM. 2005, pp. 61–70.

[JJR02] M. Jakobsson, A. Juels, and R.L. Rivest. "Making mix nets robust for electronic voting by randomized partial checking". In: *USENIX Security Symposium*. 2002, pp. 339–353.

[JK23a] Wojciech Jamroga and Yan Kim. "Practical Abstraction for Model Checking of Multi-Agent Systems". In: *Proceedings of the 20th International Conference on Principles of Knowledge Representation and Reasoning, KR 2023, Rhodes, Greece, September 2-8, 2023*. Ed. by Pierre Marquis, Tran Cao Son, and Gabriele Kern-Isberner. 2023, pp. 384–394. DOI: 10.24963/KR.2023/38. URL: https://doi.org/10.24963/kr.2023/38.

[JK23b] Wojciech Jamroga and Yan Kim. "Practical Model Reductions for Verification of Multi-Agent Systems". In: *Proceedings of the Thirty-Second International Joint Conference on Artificial Intelligence, IJCAI 2023, 19th-25th August 2023, Macao, SAR, China*. ijcai.org, 2023, pp. 7135–7139. DOI: 10.24963/IJCAI.2023/834. URL: https://doi.org/10.24963/ijcai.2023/834.

[JKK18] W. Jamroga, M. Knapik, and D. Kurpiewski. "Model Checking the SELENE E-Voting Protocol in Multi-Agent Logics". In: *Proceedings of the 3rd International Joint Conference on Electronic Voting (E-VOTE-ID)*. Vol. 11143. Lecture Notes in Computer Science. Springer, 2018, pp. 100–116.

[JKK24] Wojciech Jamroga, Yan Kim, and Damian Kurpiewski. "Scalable Verification of Social Explainable AI by Variable Abstraction". In: *Proceedings of the 16th International Conference on Agents and Artificial Intelligence, ICAART 2024*. To appear. SCITEPRESS, 2024.

[JKM20] Wojciech Jamroga, Damian Kurpiewski, and Vadim Malvone. "Natural Strategic Abilities in Voting Protocols". In: *Socio-Technical Aspects in Security and Trust - 10th International Workshop, STAST 2020, Virtual Event, September 14, 2020, Revised Selected Papers*. Vol. 12812. Lecture Notes in Computer Science. Springer, 2020, pp. 45–62. DOI: 10.1007/978-3-030-79318-0_3. URL: https://doi.org/10.1007/978-3-030-79318-0_3.

[JMP13] Hugo Jonker, Sjouke Mauw, and Jun Pang. "Privacy and verifiability in voting systems: Methods, developments and trends". In: *Computer Science Review* 10 (2013), pp. 1–30.

[JPS20] W. Jamroga, W. Penczek, and T. Sidoruk. "Strategic Abilities of Asynchronous Agents: Semantic Paradoxes and How to Tame Them". In: *CoRR* abs/2003.03867 (2020). arXiv: 2003.03867 [cs.LO]. URL: https://arxiv.org/abs/2003.03867.

[JRK22] Wojciech Jamroga, Peter Y. A. Ryan, and Yan Kim. "Verification of the Socio-Technical Aspects of Voting: The Case of the Polish Postal Vote 2020". In: (2022). DOI: 10.48550/arXiv.2210.10694. URL: https://doi.org/10.48550/arXiv.2210.10694.

[JT17] W. Jamroga and M. Tabatabaei. "Preventing Coercion in E-Voting: Be Open and Commit". In: *Electronic Voting: Proceedings of E-Vote-ID 2016*. Vol. 10141. Lecture Notes in Computer Science. Springer, 2017, pp. 1–17. DOI: 10.1007/978-3-319-52240-1_1.

[Kan+15] G. Kant, A. Laarman, J. Meijer, J. van de Pol, S. Blom, and T. van Dijk. "LTSmin: High-Performance Language-Independent Model Checking". In: *Tools and Algorithms for the Construction and Analysis of Systems. Proceedings of TACAS*. Vol. 9035. Lecture Notes in Computer Science. Springer, 2015, pp. 692–707. DOI: 10.1007/978-3-662-46681-0_61.

[Kel+10] John Kelsey, Andrew Regenscheid, Tal Moran, and David Chaum. "Attacking paper-based E2E voting systems". In: *Towards Trustworthy Elections: New Directions in Electronic Voting*. Springer, 2010, pp. 370–387.

[KJK19] Damian Kurpiewski, Wojciech Jamroga, and Michał Knapik. "STV: Model Checking for Strategies under Imperfect Information". In: *Proceedings of the 18th International Conference on Autonomous Agents and Multiagent Systems AAMAS 2019*. IFAAMAS, 2019, pp. 2372–2374.

[KJS23] Damian Kurpiewski, Wojciech Jamroga, and Teofil Sidoruk. "Towards Modelling and Verification of Social Explainable AI". In: *Proceedings of the 15th International Conference on Agents and Artificial Intelligence, ICAART 2023, Volume 1, Lisbon, Portugal, February 22-24, 2023*. Ed. by Ana Paula Rocha, Luc Steels, and H. Jaap van den Herik. SCITEPRESS, 2023, pp. 396–403. DOI: 10.5220/0011799900003393. URL: https://doi.org/10.5220/0011799900003393.

[KL07] Jonathan Katz and Yehuda Lindell. *Introduction to modern cryptography: principles and protocols*. Chapman and hall/CRC, 2007.

[KL17] P. Kouvaros and A. Lomuscio. "Parameterised Verification of Infinite State Multi-Agent Systems via Predicate Abstraction". In: *Proceedings of AAAI*. 2017, pp. 3013–3020.

[KLP04] M. Kacprzak, A. Lomuscio, and W. Penczek. "Verification of Multiagent Systems via Unbounded Model Checking". In: *Proceedings of AAMAS*. IEEE Computer Society, 2004, pp. 638–645. DOI: 10.1109/AAMAS.2004.10086.

[KR05] Steve Kremer and Mark Ryan. "Analysis of an electronic voting protocol in the applied pi calculus". In: *European Symposium on Programming*. Springer. 2005, pp. 186–200.

[KRS10] Steve Kremer, Mark Ryan, and Ben Smyth. "Election verifiability in electronic voting protocols". In: *Computer Security–ESORICS 2010: 15th European Symposium on Research in Computer Security, Athens, Greece, September 20-22, 2010. Proceedings 15*. Springer. 2010, pp. 389–404.

[KS19] Christian Killer and Burkhard Stiller. "The Swiss postal voting process and its system and security analysis". In: *International Joint Conference on Electronic Voting*. Springer. 2019, pp. 134–149.

[KSR10] Petr Klus, Ben Smyth, and Mark D Ryan. *Proswapper: Improved equivalence verifier for proverif*. 2010.

[KTV10a] R. Küsters, T. Truderung, and A. Vogt. "A game-based definition of coercion-resistance and its applications". In: *Proceedings of the 2010 23rd IEEE Computer Security Foundations Symposium*. IEEE Computer Society. 2010, pp. 122–136.

[KTV10b] Ralf Küsters, Tomasz Truderung, and Andreas Vogt. "Accountability: definition and relationship to verifiability". In: *Proceedings of the 17th ACM conference on Computer and communications security*. 2010, pp. 526–535.

[Kum+20] Ram Shankar Siva Kumar, Magnus Nyström, John Lambert, Andrew Marshall, Mario Goertzel, Andi Comissoneru, Matt Swann, and Sharon Xia. "Adversarial Machine Learning-Industry Perspectives". In: *IEEE Security and Privacy Workshops*. IEEE, 2020, pp. 69–75. DOI: 10.1109/SPW50608.2020.00028.

[Kur+21] Damian Kurpiewski, Witold Pazderski, Wojciech Jamroga, and Yan Kim. "STV+Reductions: Towards Practical Verification of Strategic Ability Using Model Reductions". In: *Proceedings of AAMAS*. ACM, 2021, pp. 1770–1772.

[KW13] S. Khazaei and D. Wikstroem. "Randomized partial checking revisited". In: *Topics in Cryptology – CT-RSA 2013*. Vol. 7779. Lecture Notes in Computer Science. Springer, 2013, pp. 115–128.

[KW19] Mazaher Kianpour and Shao-Fang Wen. "Timing Attacks on Machine Learning: State of the Art". In: *IntelliSys Volume 1*. Vol. 1037. Advances in Intelligent Systems and Computing. Springer, 2019, pp. 111–125. DOI: 10.1007/978-3-030-29516-5_10.

[LBP22] Valerio Lorenzo, Chiara Boldrini, and Andrea Passarella. *SAI Simulator for Social AI Gossiping*. https://zenodo.org/record/5780042. 2022.

[LP07] A. Lomuscio and W. Penczek. "Symbolic Model Checking for Temporal-Epistemic Logics". In: *SIGACT News* 38.3 (2007), pp. 77–99. DOI: 10.1145/1324215.1324231.

[LP08] A. Lomuscio and W. Penczek. "LDYIS: a Framework for Model Checking Security Protocols". In: *Fundamenta Informaticae* 85.1-4 (2008), pp. 359–375.

[LPQ10] A. Lomuscio, W. Penczek, and H. Qu. "Partial Order Reductions for Model Checking Temporal-epistemic Logics over Interleaved Multi-agent Systems". In: *Fundamenta Informaticae* 101.1-2 (2010), pp. 71–90. DOI: 10.3233/FI-2010-276.

[LQR10] Alessio Lomuscio, Hongyang Qu, and Francesco Russo. "Automatic Data-Abstraction in Model Checking Multi-Agent Systems". In: *Model Checking and Artificial Intelligence*. Vol. 6572. Lecture Notes in Computer Science. Springer, 2010, pp. 52–68. DOI: 10.1007/978-3-642-20674-0_4.

[LQR17] A. Lomuscio, H. Qu, and F. Raimondi. "MCMAS: An Open-Source Model Checker for the Verification of Multi-Agent Systems". In: *International Journal on Software Tools for Technology Transfer* 19.1 (2017), pp. 9–30. DOI: 10.1007/s10009-015-0378-x.

[Mar+15] T. Martimiano, E. Dos Santos, M. Olembo, and J.E. Martina. "Ceremony Analysis Meets Verifiable Voting: Individual Verifiability in Helios". In: *SECURWARE*. 2015.

[Mar20] Marshal of the Sejm (PL). *Postanowienie Marszałka Sejmu Rzeczypospolitej Polskiej z dnia 3 czerwca 2020 r. w sprawie zarządzenia wyborów Prezydenta Rzeczypospolitej Polskiej*. http://isap.sejm.gov.pl/isap.nsf/download.xsp/WDU20200000988/O/D20200988.pdf. 2020.

[McM02] K.L. McMillan. "Applying SAT Methods in Unbounded Symbolic Model Checking". In: *Proceedings of Computer Aided Verification (CAV)*. Vol. 2404. Lecture Notes in Computer Science. 2002, pp. 250–264.

[McM93] K.L. McMillan. *Symbolic Model Checking: An Approach to the State Explosion Problem*. Kluwer Academic Publishers, 1993.

[Mei+13] S. Meier, B. Schmidt, C. Cremers, and D.A. Basin. "The TAMARIN Prover for the Symbolic Analysis of Security Protocols". In: *Computer Aided Verification, Proceedings of CAV*. Vol. 8044. Lecture Notes in Computer Science. Springer, 2013, pp. 696–701. DOI: 10.1007/978-3-642-39799-8_48.

[Men09] B. Meng. "A critical review of receipt-freeness and coercion-resistance". In: *Information Technology Journal* 8.7 (2009), pp. 934–964.

[Mil71] Robin Milner. *An algebraic definition of simulation between programs*. Citeseer, 1971.

[Min20a] Minister of Infrastructure (PL). *Obwieszczenie Marszałka Sejmu Rzeczypospolitej Polskiej z dnia 18 czerwca 2020 r. w sprawie ogłoszenia jednolitego tekstu ustawy - Kodeks wyborczy*. http://isap.sejm.gov.pl/isap.nsf/download.xsp/WDU20200001319/O/D20201319.pdf. 2020.

[Min20b] Minister of Infrastructure (PL). *Rozporządzenie Ministra Infrastruktury z dnia 12 czerwca 2020 r. w sprawie przesyłek w głosowaniu korespondencyjnym*. https://isap.sejm.gov.pl/isap.nsf/download.xsp/WDU20200001037/O/D20201037.pdf. 2020.

[MN06] Tal Moran and Moni Naor. "Receipt-free universally-verifiable voting with everlasting privacy". In: *Advances in Cryptology-CRYPTO 2006*. Springer, 2006, pp. 373–392.

[Mog+14] F. Mogavero, A. Murano, G. Perelli, and M.Y. Vardi. "Reasoning About Strategies: On the Model-Checking Problem". In: *ACM Transactions on Computational Logic* 15.4 (2014), pp. 1–42.

[NV90] Rocco De Nicola and Frits W. Vaandrager. "Action versus State based Logics for Transition Systems". In: *Semantics of Systems of Concurrent Processes, Proceedings of LITP Spring School on Theoretical Computer Science*. Vol. 469. Lecture Notes in Computer Science. Springer, 1990, pp. 407–419. DOI: 10.1007/3-540-53479-2_17.

[Oka98] T. Okamoto. "Receipt-free electronic voting schemes for large scale elections". In: *Security Protocols*. Springer. 1998, pp. 25–35.

[Ott+22] Abdul-Rasheed Ottun, Pramod C. Mane, Zhigang Yin, Souvik Paul, Mohan Liyanage, Jason Pridmore, Aaron Yi Ding, Rajesh Sharma, Petteri Nurmi, and Huber Flores. "Social-aware Federated Learning: Challenges and Opportunities in Collaborative Data Training". In: *IEEE Internet Computing* (2022), pp. 1–7. DOI: 10.1109/MIC.2022.3219263.

[Pal+23a] Luigi Palmieri, Chiara Boldrini, Lorenzo Valerio, Andrea Passarella, and Marco Conti. "Exploring the Impact of Disrupted Peer-to-Peer Communications on Fully Decentralized Learning in Disaster Scenarios". In: *International Conference on Information and Communication Technologies for Disaster Management, ICT-DM*. IEEE, 2023, pp. 1–6. DOI: 10.1109/ICT-DM58371.2023.10286953.

[Pal+23b] Luigi Palmieri, Lorenzo Valerio, Chiara Boldrini, and Andrea Passarella. "The effect of network topologies on fully decentralized learning: a preliminary investigation". In: *CoRR* abs/2307.15947 (2023). DOI: 10.48550/ARXIV.2307.15947. arXiv: 2307.15947.

[Pańa] Państwowa Komisja Wyborcza [translated: National Electoral Commission]. *Kodeks Wyborczy [translated: Electoral Code]*. URL: https://pkw.gov.pl/prawo-wyborcze/kodeks-wyborczy (visited on 2023-03-31).

[Pańb] Państwowa Komisja Wyborcza [translated: National Electoral Commission]. *Wybory Prezydenta Rzeczypospolitej Polskiej 2010 r.* URL: https://prezydent2010.pkw.gov.pl/ (visited on 2023-03-31).

[Pańc] Państwowa Komisja Wyborcza [translated: National Electoral Commission]. *Wybory Prezydenta Rzeczypospolitej Polskiej 2015 r.* URL: https://prezydent2015.pkw.gov.pl/ (visited on 2023-03-31).

[Pańd] Państwowa Komisja Wyborcza [translated: National Electoral Commission]. *Wybory Prezydenta Rzeczypospolitej Polskiej 2020 r.* URL: https://prezydent20200628.pkw.gov.pl/ (visited on 2023-03-31).

[Pań20a] Państwowa Komisja Wyborcza [translated: National Electoral Commission]. *Uchwała nr 167/2020 PKW z dnia 8 czerwca 2020 r. w sprawie sposobu postępowania z kopertami zwrotnymi i pakietami wyborczymi w głosowaniu korespondencyjnym w kraju*. https://pkw.gov.pl/uploaded_files/1591677157_uchwala-w-sprawie-sposobu-postepowania-z-kopertami-zwrotnymi.pdf. 2020.

[Pań20b] Państwowa Komisja Wyborcza [translated: National Electoral Commission]. *Uchwała nr 182/2020 PKW z dnia 10 czerwca 2020 r. w sprawie określenia wzoru i rozmiaru koperty na pakiet wyborczy, koperty zwrotnej, koperty na kartę do głosowania, oświadczenia o osobistym i tajnym oddaniu głosu na karcie do głosowania oraz instrukcji głosowania korespondencyjnego w wyborach Prezydenta Rzeczypospolitej Polskiej zarządzonych na dzień 28 czerwca 2020 r.* https://pkw.gov.pl/uploaded_files/1591872974_uchwala-nr-182-2020-pkw.pdf. 2020.

[Pań20c] Państwowa Komisja Wyborcza [translated: National Electoral Commission]. *Ustawa z dnia 2 czerwca 2020 r. o szczególnych zasadach organizacji wyborów powszechnych na Prezydenta Rzeczypospolitej Polskiej zarządzonych w 2020 r. z możliwością głosowania korespondencyjnego [translated: The Act of 2 June 2020 on special rules for the organisation of general elections of the President of the Republic of Poland ordered in 2020 with the possibility of postal voting]*. https://prezydent20200628.pkw.gov.pl/prezydent20200628/statics/prezydent_20200628_kodeks/uploaded_files/1591242112_ustawa-organizacja-wyborow-w-2020.pdf. 2020.

[Pel93] Doron A. Peled. "All from One, One for All: on Model Checking Using Representatives". In: *Proceedings of CAV*. Ed. by Costas Courcoubetis. Vol. 697. Lecture Notes in Computer Science. Springer, 1993, pp. 409–423. DOI: 10.1007/3-540-56922-7_34.

[PL03] W. Penczek and A. Lomuscio. "Verifying Epistemic Properties of Multi-Agent Systems via Bounded Model Checking". In: *Proceedings of AAMAS*. Melbourne, Australia: ACM Press, 2003, pp. 209–216. ISBN: 1-58113-683-8.

[Pri83] L. Priese. "Automata and Concurrency". In: *Theoretical Computer Science* 25 (1983), pp. 221–265. DOI: 10.1016/0304-3975(83)90113-5.

[PS15] D. Pattinson and C. Schürmann. "Vote Counting as Mathematical Proof". In: *Advances in Artificial Intelligence, Proceedings of AI*. Vol. 9457. Lecture Notes in Computer Science. Springer, 2015, pp. 464–475. DOI: 10.1007/978-3-319-26350-2_41.

[Riv06] R. Rivest. "The ThreeBallot Voting System". Available online at http://theory.csail.mit.edu/~rivest/Rivest-TheThreeBallotVotingSystem.pdf. 2006.

[RRI16] P.Y.A. Ryan, P.B. Rønne, and V. Iovino. "Selene: Voting with Transparent Verifiability and Coercion-Mitigation". In: *Financial Cryptography and Data Security: Proceedings of FC 2016. Revised Selected Papers*. Vol. 9604. Lecture Notes in Computer Science. Springer, 2016, pp. 176–192. DOI: 10.1007/978-3-662-53357-4_12.

[RST15] Peter Y. A. Ryan, Steve A. Schneider, and Vanessa Teague. "End-to-End Verifiability in Voting Systems, from Theory to Practice". In: *IEEE Security & Privacy* 13.3 (2015), pp. 59–62. DOI: 10.1109/MSP.2015.54.

[RT13] P.Y.A. Ryan and V. Teague. "Pretty Good Democracy". In: *Security Protocols XVII*. Vol. 7028. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2013, pp. 111–130.

[Rya+09] P.Y.A Ryan, D. Bismark, J. Heather, S. Schneider, and Z. Xia. "Prêt à voter: a voter-verifiable voting system". In: *Information Forensics and Security, IEEE Transactions on* 4.4 (2009), pp. 662–673.

[Rya10] P.Y.A. Ryan. "The Computer Ate My Vote". In: *Formal Methods: State of the Art and New Directions*. Springer, 2010, pp. 147–184.

[Sch03] Ph. Schnoebelen. "The Complexity of Temporal Model Checking". In: *Advances in Modal Logics, Proceedings of AiML 2002*. World Scientific, 2003.

[Sch04] Pierre-Yves Schobbens. "Alternating-time logic with imperfect recall". In: *Electronic Notes in Theoretical Computer Science* 85.2 (2004), pp. 82–93.

[Sej22] Sejm Rzeczypospolitej Polskiej. *Internetowy System Aktów Prawnych [translated: Internet Legal Acts System]*. https://isap.sejm.gov.pl/isap.nsf/search.xsp?status=O&kw=wybory. 2022.

[SG04] Sharon Shoham and Orna Grumberg. "Monotonic Abstraction-Refinement for CTL". In: *Proceedings of TACAS*. Vol. 2988. Lecture Notes in Computer Science. Springer, 2004, pp. 546–560. DOI: 10.1007/978-3-540-24730-2_40.

[SK95] Kazue Sako and Joe Kilian. "Receipt-free mix-type voting scheme: A practical solution to the implementation of a voting booth". In: *Advances in Cryptology—EUROCRYPT'95: International Conference on the Theory and Application of Cryptographic Techniques Saint-Malo, France, May 21–25, 1995 Proceedings 14*. Springer. 1995, pp. 393–403.

[Sku20] Marcin Skubiszewski. *Obserwatorium Wyborcze do Przewodniczącego PKW: Nieprawidłowo wydrukowane karty do głosowania za granicą - konieczność rozwiązania problemu*. 2020-06-22. URL: https://monitorkonstytucyjny.eu/archiwa/14355 (visited on 2022-04-27).

[SL09] Y. Shoham and K. Leyton-Brown. *Multiagent Systems - Algorithmic, Game-Theoretic, and Logical Foundations*. Cambridge University Press, 2009. ISBN: 978-0-521-89943-7.

[Soc22] Social AI gossiping. Micro-project in Humane-AI-Net. *Project website*. https://www.ai4europe.eu/research/research-bundles/social-ai-gossiping. 2022.

[Soc24] Social Explainable AI, CHIST-ERA. *Project website*. http://www.sai-project.eu/. 2021–2024.

[Spo20] Spotted Lublin editors. *Wybory 2020. Karty do głosowania bez czerwonej pieczęci obwodowej komisji wyborczej*. 2020-06-29. URL: https://spottedlublin.pl/wybory-2020-karty-do-glosowania-bez-czerwonej-pieczeci-obwodowej-komisji-wyborczej/ (visited on 2022-05-14).

[Sti05] Douglas R Stinson. *Cryptography: theory and practice*. Chapman and Hall/CRC, 2005.

[SV20] Diego Sempreboni and Luca Vigano. "X-Men: A mutation-based approach for the formal analysis of security ceremonies". In: *2020 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE. 2020, pp. 87–104.

[TJR16] M. Tabatabaei, W. Jamroga, and Peter Y. A. Ryan. "Expressing Receipt-Freeness and Coercion-Resistance in Logics of Strategic Ability: Preliminary Attempt". In: *Proceedings of the 1st International Workshop on AI for Privacy and Security, PrAISe@ECAI 2016*. ACM, 2016, 1:1–1:8. DOI: 10.1145/2970030.2970039.

[Top+21] Mustafa Toprak, Chiara Boldrini, Andrea Passarella, and Marco Conti. "Harnessing the Power of Ego Network Layers for Link Prediction in Online Social Networks". In: *CoRR* abs/2109.09190 (2021). arXiv: 2109.09190.

[Uni48] United Nations. *Universal Declaration of Human Rights*. 1948-12. URL: https://www.un.org/sites/un2.un.org/files/2021/03/udhr.pdf.

[Upp02] Uppsala University and Aalborg University. *UPPAAL documentation*. Version 4.1.x. 2002. URL: https://docs.uppaal.org/.

[Wei99] G. Weiss, ed. *Multiagent Systems. A Modern Approach to Distributed Artificial Intelligence*. MIT Press: Cambridge, Mass, 1999.

[Woo02] M. Wooldridge. *An Introduction to Multi Agent Systems*. John Wiley & Sons, 2002.

[Zol20] Marie-Laure Zollinger. *From secure to usable and verifiable voting schemes*. 2020.

[ZRR20] M-L. Zollinger, P. Roenne, and P.Y.A. Ryan. "Mechanized Proofs of Verifiability and Privacy in a paper-based e-voting Scheme". In: *Proceedings of 5th Workshop on Advances in Secure Electronic Voting*. 2020.