Automatic verification of multi-agent systems security properties specified with LTL

Kholud Alghamdi and Marius Silaghi
Florida Institute of Technology
kalghamdi2017@my.fit.edu, msilaghi@fit.edu

Abstract

We propose a way to verify security requirements of critical multi-agent system processes by using logic representations and automatic reasoning. The typical multi-agent system considered in our work would be an election system with agents representing their users and aiming to ensure security. Relevant processes are authentication, voting, re-voting, and election verification. The security requirements commonly addressed in such a voting system are: no user can vote unless it got authenticated, no invalid vote should be counted, no vote should be counted twice, and each valid vote should be eventually counted. We show a model of such security requirements by using system liveness properties, and exemplify their verification on a real system that we implement for this purpose.

Introduction

We exemplify how logic representations can be used to verify security requirements of multi-agent system processes, in particular, in the frame of voting systems.

The approach followed uses formal verification and logic representation to check security requirements with respect to the behavior/functionality of the main internal processes in a voting system. The method consists of three phases. A modeling phase which refers to modeling both the internal system processes and desirable security requirements. The second phase, also referred to as the Running Phase, consists of using a model checking technique to verify the validity of the properties of the model, and eventually to find any counterexample, if one exists. The last phase, called the Analysis Phase, uses the result of the Running Phase to either accept the design or to change it and return to phase one.

In order to evaluate the proposed method's potential for verifying security requirements we implement a realistic voting system as reference. The main processes supported by this voting system are: authentication, voting, and verification of election. We model these processes individually. To also model the security requirements, we start by defining them unambiguously in natural language. Based on this definition, a Linear Temporal Logic (LTL) specification is formulated. All LTL formulas obtained are converted to non-deterministic Büchi automata (BA) models (Richard and others 1962; Gastin and Oddoux 2001), namely models based on non-deterministic finite state machines with transitions on inputs and that accept sequences with infinitely occurring states respecting the acceptance condition.

We here observe that security requirements can be modeled with liveness properties that map easily into such acceptance conditions.

This study shows that security requirements of the internal multi-agents system processes can be tested based on liveness system properties when they are modelled with Linear Temporal Logic representations.

This paper is structured as follows. After the background and related work is detailed, the proposed illustrative models are formalized, and the experiment is detailed. We conclude after the analysis of the experiment.

Background

Model checking is a research area concerned with verifying whether the model of a system satisfies given specifications or not. It requires two things; a mathematical (or an abstract) model of the system under test and specifications of the desired behaviors or requirements. By specifications one refers to formal definitions of properties we want the model to satisfy. Some properties can be described as invariants. If the invariant is not violated, that means it holds for infinite many states in the future of the model evolution. If there is a single violation of a given property, then the counterexample can be found (Clarke Jr et al. 2018).

A mathematical model of a system in this context refers to the specification of system state transitions as a tuple $T = \langle S, I, A, \delta, AP, L \rangle$, where:

- S is the set of possible states,
- I is the initial state distribution,
- A is a set of inputs,
- δ is a relation modelling potentially non-deterministic transitions between any two states given an input,
- AP is a set of atomic propositions which are either true or false at any given time,
- L is a labeling function telling which members of AP hold true for each state.

The automata defined by T accepts inputs from A as specified in δ and non-deterministically produces sequences of

states, which by virtue of L translate into sequences of corresponding sets of atomic propositions from AP, denoted Ω (Müller-Olm, Schmidt, and Steffen 1999).

An abstract model (Giunchiglia, Villafiorita, and Walsh 1997) of a system in this context refers to representation with a language that skips certain details.

Linear Temporal Logic A logic system commonly used to reason with complex dynamic systems is the Linear Temporal Logic (LTL) (Rescher and Urquhart 1971; Francez and Pnueli 1978; Rozier 2011).

LTL is based on a finite set of atomic propositions, can be equipped with logical or temporal operators such as the temporal connectives: **until** denoted U, **release** denoted R, **next** denoted X or $\bigcirc$, **globally** or always in the future denoted G or $\Box$. **next** X π means that π is true in the next time step after the current one. $\Diamond$ means eventually will happen in the future and $\blacklozenge$ means eventually happened at some time in the past.

For example, π U Ψ means that either Ψ is true now or π is true now and π remains true at least until Ψ holds. On the other hand, π R Ψ means π releases Ψ, specifies that Ψ should be true now and remains true until π is true, inclusively.

Furthermore, LTL is commonly used for formally describing properties of dynamic systems. It allows representing many real verbal English language requirements with unambiguous logic expressions.

The basis of LTL is propositional logic, with the main difference lying in the addition of temporal operators. Temporal logic allows for making deductive arguments about not only what is, but what was, what will be, what has always been and what always will be.

Common LTL-provable attributes of a system are phrased as liveness, safety, correctness, or invariants (Rescher and Urquhart 1971). Liveness is the property stating that a given predicate keeps happening past any specific time in the future.

An LTL property ϕ can be used to specify a system requirement such that it can be formally proven. The main goal of model checking is to analyze the system model when reasoning about properties of infinite sequences of states it can generate.

Every valid LTL formula can be translated to a corresponding Büchi Automaton that accepts all and only the infinite traces that satisfy the formula (Vardi and Wolper 1986). The Büchi automaton obtained by conversion from an LTL formula ϕ is denoted by $B\ \phi$.

To show that a system model satisfies an LTL formula ϕ, this formula is commonly converted to a Büchi Automata.

Büchi Automata Büchi Automata are often used in model checking as an automata-theoretic version of a formula in linear temporal logic. According to (Richard and others 1962; Gastin and Oddoux 2001) a Büchi automaton is a finite automaton and for non-deterministic systems it can be defined as $\mathcal{A} = (Q, \Sigma, \delta, I, F)$ where:

- Q is a finite set of states
- Σ is a finite set of inputs, (aka. the alphabet)
- $\delta \subseteq Q \times \Sigma \times Q$, is called the transition relation
- $I \subseteq Q$ is the set of initial states
- $F \subseteq Q$ is the set of accepting states

Static code analysis is an automatic program checking process commonly implemented with two components: an automaton and the inputs for this automaton. It requires to extracts the input from the source code of the system that needs to be verified, and feed it into the automaton. It requires that the source code is first translated into an intermediate model for analysis and a call graph is generated from the source code.

Each path contained in the call graph is considered as an input path for the automaton for the property checking.

To start the process of model checking, the ϕ must be converted to $B\ \phi$ such that the inputs to the $B\ \phi$ are from A as specified in δ. If all the inputs are accepted in $B\ \phi$, then we can conclude that the model satisfies the property by the definition of BA. If there is at least one counterexample that is not accepted in $B\ \phi$, then the model is not satisfying the property.

LTL2BA (Gastin and Oddoux 2001) is an available software used to translate the LTL formulas into Büchi automaton and it draws automatically the resulting automaton and generates the PROMELA codes which can be given to a model checker to verify properties on a system.

PROMELA stands for Process meta language, and it is composed of global declarations such as shared variables, communication channels, and process types. PROMELA is a language used to build verification models. The verification model represents the abstraction of a design and contains only the properties relevant for the desired verification.

Models

In our approach, the internal processes are modelled individually. These are: the authentication process, the voting process, and the election verification process.

We make use of temporal descriptions in security requirements, such as "the voter should be successfully authenticated as a precondition of casting the vote", "a vote should be eventually counted after it is cast", "a voter should eventually receive a confirmation message after he/she has voted". We utilize words like "precondition", "before", and "after" to show that some actions will or should happen in some sequence.

We note that types of temporal descriptions occurring in our cases of interest can be formulated as liveness properties. Liveness properties state that a given predicate eventually holds at moments past any specific time in the future.

To illustrate the approach, in the following, we used linear temporal logic to formulate liveness properties describing system security requirements:

- Formula 1: $\Box$ (Signin (Valid-Username, Valid-Password) $\rightarrow \diamond$ Verified (Valid-Username, Valid-Password))

- Formula 2: □ (Verified (Valid-Username, Valid-Password) → ⋄ Receive (Unique-Code))
- Formula 3: □ (Cast (vote) → ⋄ (Counted (vote) ∧ Confirmed(vote)))

The current working assumptions, that are included as limitations in the attacker model of this research step, state that computer hardware will not fail, Internet and SMS communication is reliable and private, and human participants will physically be unconstrained and actively participating during the whole process. Relaxations of these assumptions are subject of future work.

The first aforementioned sample LTL statement specifies that whenever a sign-in action happens for a valid user, then sometime later there will be a verification action taking place and qualifying him. Further, according to the second statement, whenever the user is verified, then sometime later he receives a unique-code as a token for casting a vote. Third, whenever a vote is cast, then sometime later the confirmation message must be generated and the vote is counted.

After that, tools such as LTL2BA are found to be able to create the Büchi automaton from such LTL properties. The automaton generated in this way is specified in PROMELA.

The call graph of the source code is built and it is focused only on function calls, then execution paths of the implementation are also provided as the traversal from root to end nodes.

For example, verification pseudo code corresponding to the automata generated for the first LTL property is shown in Algorithm 1.

```
signin ();
getVerified ();
signin-completion ();
while ( valid-username, valid-password ) do
  getVerified ();
```

Algorithm 1: Pseudo-code Property I

As shown in the Pseudo-code, the function names are: { signin, getVerified, signin-completion }

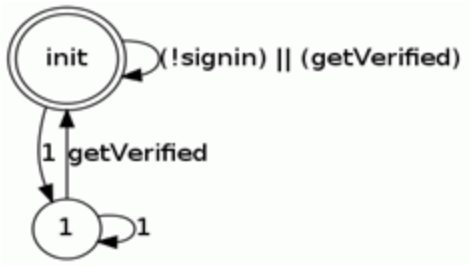
The path illustrated in Algorithm 1 is extracted from the call graph of these functions.

The Büchi automata would be fed by the set of this paths as inputs. Then, if the paths from the input set end in the accepting state, which is the "init" state, the property is achieved. In contrast, if the paths end up in a non-accepting state "1", then the property is not guaranteed.

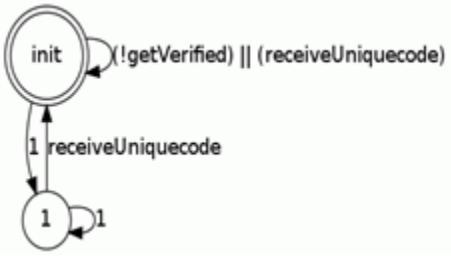
Experiment

The proposed process is experimented with a voting system developed for Saudi Arabia's business performance contests organized under the public administration.

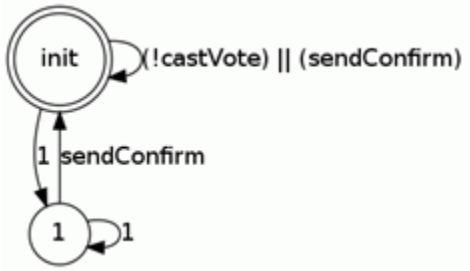
The aforementioned formulas ϕ are translated into a Buchi automaton B ϕ as shown in the next figures generated automatically by LTL2BA.



"!signin" means any event but signin, and (| |) is utilized to match either not signin or getVerified.



"!getVerified " means any event but getVerified, and (| |) is utilized to match either not getVerified or receiveUniqueCode.



"!castVote " means any event but castVote, and (| |) is utilized to match either not castVote or sendConfirm.

In each of these examples there are two states. As named by the LTL2BA tool generating the automata, the "init" state is the accepting state while both states of the automaton can be initial states. The transition labels are the predicate names. Label "1" matches any predicate name.

For each BA, a PROMELA code is generated, as shown below:

First Büchi Automata Code

```
never {
 /* □ (signin → ◊ getVerified) */
accept-init : /* init */
if
:: (!signin) | | (getVerified) → goto accept-init
:: (1) → goto T0-S2
fi;
T0-S2 : /* 1 */
if
:: (getVerified) → goto accept-init
:: (1) → goto T0-S2
fi;
}
```

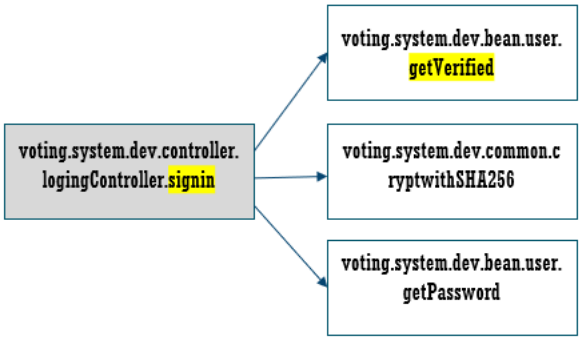
Second Büchi Automata Code

never {
/* □ (getVerified → ◊ receiveUniqueCode) */
accept-init : /* init */
if
:: (!getVerified) | | (receiveUniqueCode) → goto accept-init
:: (1) → goto T0-S2
fi;
T0-S2 : /* 1 */
if
:: (receiveUniqueCode) → goto accept-init
:: (1) → goto T0-S2
fi;
}

Third Büchi Automata Code

never {
/* □ (castVote → ◊ sendConfirm) */
accept-init : /* init */
if
:: (!castVote) | | (sendConfirm) → goto accept-init
:: (1) → goto T0-S2
fi;
T0-S2 : /* 1 */
if
:: (sendConfirm) → goto accept-init
:: (1) → goto T0-S2
fi;
}

For example, a fragment of the function call graph corresponding to the authentication process in the source code is shown below.



The extracted path from the source code of the authentication process is: { signin, getVerified, signin-completion }

A sample input for the first Büchi automaton is the set of path: { {signin, getVerified, signin-completion, getVerified}; {signin, getVerified, signin-completion, getVerified}; {signin, getVerified, signin-completion, getVerified, getVerified } }

As noted, all input sets end in "init"' state which is the accepting state, therefore the property is achieved with this sample.

Conclusion

Important processes like voting and its verification are frequently implemented by programmers and are known to be error prone and be subject to many security challenges. Automatic verification is proposed as a design and implementation step to increase robustness. Security requirements modeled using Liner Temporal Logic (LTL) are verified directly using the source code.

It is showed that the security requirements of the internal multi-agents system processes can be tested based on liveness system properties when they are modelled with linear temporal logic representations.

It is observed that all the identified security properties that we wanted to pose at this stage were naturally implemented in LTL. The concept was tested on a sample voting system developed for Saudi business performance contests.

Properties in LTL are further translated into Buchi automata (BA). We present such translations using Buchi automata, then write the pseudo-code based on the automata PROMELA code. We showed that the corresponding properties of the security requirements can be verified with the automatic system. Thus a step is achieved towards the use of artificial intelligence for guaranteeing security properties in election systems.

References

Clarke Jr, E. M.; Grumberg, O.; Kroening, D.; Peled, D.; and Veith, H. 2018. *Model checking*. MIT press.

Francez, N., and Pnueli, A. 1978. A proof method for cyclic programs. *Acta Informatica* 9(2):133–157.

Gastin, P., and Oddoux, D. 2001. Fast ltl to büchi automata translation. In *International Conference on Computer Aided Verification*, 53–65. Springer.

Giunchiglia, F.; Villafiorita, A.; and Walsh, T. 1997. Theories of abstraction. *AI communications* 10(3, 4):167–176.

Müller-Olm, M.; Schmidt, D.; and Steffen, B. 1999. Model-checking. In *International Static Analysis Symposium*, 330–354. Springer.

Rescher, N., and Urquhart, A. 1971. Temporal logic springer verlag. *Vienna-New York*.

Richard, B., et al. 1962. On a decision method in restricted second order arithmetic. In *Proc. of the International Congress on Logic, Method and Philosophy of Science, 1962*, 425–435. Stanford University Press.

Rozier, K. Y. 2011. Survey: Linear Temporal Logic Symbolic Model Checking. *Computer Science Review* 5(2):163–203.

Vardi, M. Y., and Wolper, P. 1986. An automata-theoretic approach to automatic program verification. In *Proceedings of the First Symposium on Logic in Computer Science*, 322–331. IEEE Computer Society.