Squeezing State Spaces of (Attack–Defence) Trees

Conference Paper · November 2019

DOI: 10.1109/ICECCS.2019.00015

CITATIONS
0

READS
39

4 authors:

Laure Petrucci
Université Paris 13 Nord
159 PUBLICATIONS **1,849** CITATIONS
SEE PROFILE

Michał Knapik
Polish Academy of Sciences
26 PUBLICATIONS **88** CITATIONS
SEE PROFILE

Wojciech Penczek
Polish Academy of Sciences
220 PUBLICATIONS **2,309** CITATIONS
SEE PROFILE

Teofil Sidoruk
Polish Academy of Sciences
3 PUBLICATIONS **1** CITATION
SEE PROFILE

Squeezing State Spaces of (Attack-Defence) Trees

Laure Petrucci
LIPN, CNRS UMR 7030,
Université Paris 13, Sorbonne Paris Cité
99 av. J-B. Clément, 93430 Villetaneuse, France
laure.petrucci@lipn.univ-paris13.fr

Michał Knapik, Wojciech Penczek, Teofil Sidoruk[1]
Institute of Computer Science
Polish Academy of Sciences
Jana Kazimierza 5, 01-248 Warsaw, Poland
{*m.knapik, w.penczek, t.sidoruk*}*@ipipan.waw.pl*

***Abstract*—In earlier work, we presented translations of attack-defence trees (ADTrees) to extended asynchronous multi-agent systems. By avoiding some sequences, agent models constructed via these transformations already embed state space reductions. Here, we introduce Guarded Update Systems and their synchronisation topology, allowing us to define a new general reduction scheme that applies to tree topologies, and in particular to ADTrees. The reduction exploits the layered structure of a tree by avoiding unnecessary interleavings between nodes at different depths. We prove the soundness of this new method and present extensive experimental results, including scalable models, to demonstrate it can be effectively used alongside previously employed techniques.**

1. Introduction

Attack-defence trees [1], [2] have been developed as an extension of attack trees [3], [4], [5], [6], [7], [8] to study interactions between attacker and defender parties. They provide a simple graphical formalism enjoying advanced modelling versatility. The paper [9] models attack-defence trees in an agent-aware formalism in order to analyse the impact of different agents distributions over the tree nodes, i.e. which agent performs which task for which goal. It turns out that this determines not only the *performance* but also the *feasibility* of an attack or defence strategy, and permits quantifying performance metrics (e.g. cost and time) of attack/defence strategies under distinct agents coalitions. Using the tool IMITATOR [10], we determined the feasibility of coalition strategies, and synthesised the value of the attributes that make them feasible.

In this paper we recall the translation of attack-defence trees (ADTrees) to extended multi-agent systems [9], which was defined in such a way that some sequences of actions were avoided in order to construct models already reduced. Moreover, we introduce Guarded Update Systems and a tree synchronisation topology, allowing for defining a new general reduction scheme that applies in particular to ADTrees. The reduction exploits the layered structure of a tree by avoiding unnecessary interleavings between nodes at different depths. Since for a node to complete its descendants should have completed as well, we introduce a variable counting the nodes having already completed at a given depth. In this new reduction scheme, the nodes at depth n synchronise with their children but wait for all nodes at depth $n+1$ to complete before proceeding further, and not only for their children. This allows for avoiding interleavings between actions of nodes at depth n with those of children of other nodes at depth $n+1$, and leads to an efficient state space reduction preserving the desired properties. We prove the soundness of this new method and present extensive experimental results, including scalable models, to demonstrate it can be effectively used alongside previously employed techniques.

Contributions. In this paper we introduce: (i) a definition of Guarded Update Systems and a tree synchronisation topology, (ii) a definition of a novel general reduction scheme that applies in particular to ADTrees, (iii) a proof of the soundness of this new method, (iv) measurements of the impact of the novel reduction on model sizes, exercised on several scalable models.

Related Work. The original formalism of *Attack-Defence Trees* [1], [11] has been implemented in several analysis frameworks based on Timed Automata [12], I/O-IMCs [13], Bayesian Networks [14], stochastic games [15], etc. Since each framework computes queries for the specific focus of the underlying semantic model, it is hard (or sometimes even infeasible) to translate ADTree models and queries among the frameworks. In [9] the work of [16] was extended to ADTrees using Extended Asynchronous Multi-Agent Systems (EAMAS for short) and *synthesising* (constraints for the) values of attributes that yield the effectiveness of an attack or defence. Moreover, the EAMAS semantic formalism offers a succinct representation amenable to efficient state-space reduction techniques, which helps deploying lightweight analyses in comparison to other highly expressive formalisms, such as Attack-Defence Diagrams [17]. In this paper we show how to define a new general reduction scheme for tree topologies that applies in particular to ADTrees.

This work was supported by the PICS CNRS/PAN project PARTIES and by PAN. Michał Knapik is supported by POLLUX VoteVerif project.
[1]*Also affiliated with Faculty of Mathematics and Information Science, Warsaw University of Technology, Koszykowa 75, 00-662 Warsaw, Poland*

Outline. The paper is structured as follows. In Section 2 we recall the ADTree model. Section 3 defines networks of Guarded Update Systems and a compositional, executable semantics for ADTrees in terms of these. Section 4 deals with a tree structure of synchronisation topology and shows how it can be exploited to obtain a reachability-preserving reduction of the system. The effectiveness of our approach is demonstrated in Section 5, where we present extensive experimental results obtained on scalable models. This work concludes in Section 6, where we finally draw possible lines of future research.

2. Attack-Defence Trees

Attack trees are well-known tree representations of attack scenarios, that allow for evaluating the security of complex systems to the desired degree of refinement [3]. The root of the tree is the attacker's goal, and the children of a node represent refinements of the node's goal into subgoals. The tree leaves are (possibly quantified) foreseeable attacker's actions. *Attack-defence trees* (ADTrees, [1]) are an extension of attack trees including possible counteractions of a defender, thus representing security scenarios as an interplay between attacker's and defender's actions. This can model mechanisms triggered by the occurrence of specified opposite actions.

Countering actions express pointwise interactions between opposite players. This can be reactive, such as the police in Example 1 (see further), or passive, such as message encryption to avoid eavesdropping in open channels. The intuition behind the counter actions in [1] can be described as follows:

- *counter defence* $A = \texttt{CAND}(a, d)$: if attack a succeeds and the countering defence d fails, then the attack goal A is successful;
- *counter attack* $D = \texttt{CAND}(d, a)$: if defence d happens and attack a fails, then the defence goal D is successful.

A selection of various types of nodes composing ADTrees is shown in Table 1. As with counter defence/attack above, these constructs are symmetric for attack or defence goals. In the graphical representations, triangular nodes indicate (arbitrary) subtrees, whereas circular (resp. rectangular) nodes represent attack (resp. defence) leaves.

In Table 1, the operators for a *choice* between a successful attack and a failing defence (named *no defence*), and vice-versa (*inhibiting attack*) are less usual constructs in the literature [6], yet they model realistic scenarios such as attack goals succeeding by security negligence rather than by performing an elaborated (and costly!) attack. This is of interest for quantitative analyses of e.g. cost and probability.

The survey of [6] lists several ways to model order-dependent events: sequential enforcement, time dependent ordered-AND, priority- or sequential-AND, etc. For attack trees, in the sequentially ordered conjunction (SAND) of [5], attacks succeed when all children occurred in the required order. This abstracts away the exact time of occurrence of events; with few exceptions, e.g. [18], this is the preferred approach in the literature [4], [5], [6], [7], [8]. Therefore SANDs describe the order in which actions *will* take place, rather than a potential order of events. Their role is to enforce *sequential events* and rule out parallel executions.

The paper [9] also introduced SCAND gates: sequential gates that have attacks and defences as children. They allow for representing a successful attack followed by a failed defence (named *failed reactive defence* in Table 1), and vice versa. SCAND gates model an attack goal that must overcome some reactive defence, triggered only after the incoming attack has been detected.

Attributes of ADTrees. Attributes [2], [4], [7] are numeric properties of attack/defence nodes that allow for quantitative analyses. Typical attributes include cost, time, and probability. As in [16], [9], attributes can be parameters over which we synthesize constraints indicating which attribute values can lead to a successful attack.

Attributes concern all constructs (i.e. nodes) in ADTrees, because a construct may not be fully described by just its children. An attribute is then given by a node's *initial value*, and a *computation function*.

In [9], each action described by an ADTree construct can be performed by a particular *agent*. Different attacks/defences could be handled by one or multiple agents, which allows to express properties on agents coalitions. Each node in the ADTree is assigned to an agent, and a single agent can handle multiple nodes. In the general case, the only constraint is that no agent handles both attack and defence nodes. As shown above, specific ADTrees might enforce additional constraints. In this paper, we do not consider agents, that are left to future work.

Conditional counter measures. It may happen that a countering node has a successful or unsuccessful outcome depending on the attributes of its children. Therefore, *conditions* may be associated with countering nodes, which are Boolean functions over the attributes of the ADTree. When present, the condition then comes as an additional constraint for the node operation to be successful.

Example 1 (from [9])**.** *We present a simple example in Figure 1, featuring thieves that try to steal a treasure in a museum. To achieve their goal, they first must access the treasure room, which involves distracting and bribing a guard (*b*), and forcing the secure door (*f*). Both actions are costly and take some time.*

*After these actions succeed the thieves can steal the treasure (*ST*), which takes a little time for opening its display stand and putting it in a bag. Then the thieves are ready to flee (*TF*), but to do that they have to choose an escape route to go away (*GA*): this can be a spectacular escape in a helicopter (*h*), or a mundane escape via the emergency exit (*e*). The helicopter is expensive but fast while the fire exit is slower but at no cost. These values will allow for experimenting different solutions. Furthermore, the time to perform a successful escape can be inversely proportional to the number of agents involved in the robbery. This is easy to encode via a computation function in the gate* GA*.*

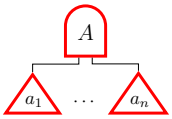
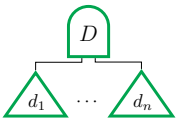
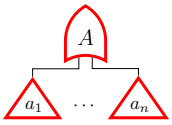
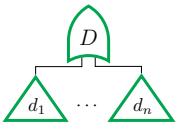
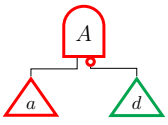
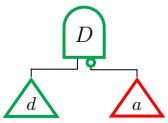
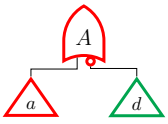
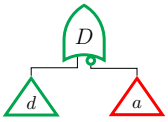
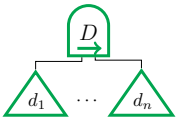
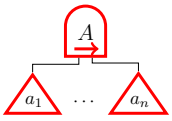
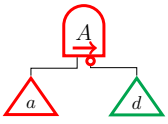
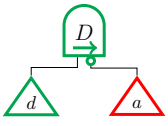
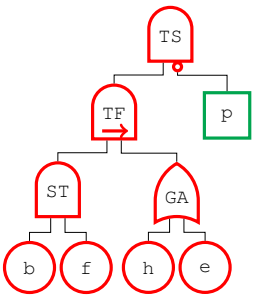
Name	Graphics
Attack	
Defence	
And attack	
And defence	
Or attack	
Or defence	
Counter defence	
Counter attack	
No defence	
Inhibiting attack	
Sequential and defence	
Sequential and attack	
Failed reactive defence	
Sequential counter attack	

Table 1: ADTree constructs

Figure 1: The treasure hunters



(a) ADTree

Name	Cost	Time
TS (treasure stolen)		
p (police)	100 €	10 min
TF (thieves fleeing)		
ST (steal treasure)		2 min
b (bribe gatekeeper)	500 €	1 h
f (force arm. door)	100 €	2 h
GA (get away)		
h (helicopter)	500 €	3 min
e (emergency exit)		10 min

Condition for TS:
$init\_time(\texttt{p}) > init\_time(\texttt{ST}) + time(\texttt{GA})$

(b) Attributes of nodes

As soon as the treasure room is penetrated (i.e. after b *and* f *but before* ST*) an alarm goes off at the police station, so while the thieves flee the police hurries to intervene (*p*). The treasure is then successfully stolen iff the thieves have fled and the police failed to arrive or does so too late. This last possibility is captured by the additional condition associated with the treasure stolen gate (*TS*), which states that the arrival time of the police must be greater than the time for the thieves to steal the treasure and go away. The minimum time for a successful attack is 3 h 5 min for a single agent, and 2 h 5 min for coalitions of two or more agents. In both cases, the minimum cost is 1100€.*

3. ADTree Semantics and $\mathcal{GUS}$s

In this section we recall a compositional, executable semantics for ADTrees [9], but in terms of their translation into networks of Guarded Update Systems ($\mathcal{GUS}$s). The networks of $\mathcal{GUS}$s are similar to Extended Asynchronous Multi-Agent Systems of [9], but simpler as they do not cater for agents. $\mathcal{GUS}$s that exhibit a tree topology allow for defining a state space reduction, which can be applied to ADTrees in particular.

3.1. Guarded Update Systems

Guarded Update Systems are simply automata with variables and guarded transitions. Although several tools using automata, e.g. UPPAAL or IMITATOR, allow for manipulating variables, we formally define such automata with some restrictions that avoid conflicts in updates.

Definition 1 (Guarded Update Systems)**.** *Let $Vars$ be a finite set of integer variables. A* Guarded Update System *($\mathcal{GUS}$) is a tuple $\mathcal{M} = \langle \mathcal{S}, s^0, \rightarrow, Acts \rangle$ where:*

1) *$\mathcal{S}$ is a finite set of states and $s^0 \in \mathcal{S}$ the initial state;*
2) *$\rightarrow \subseteq \mathcal{S} \times Acts \times G \times U \times \mathcal{S}$ is a transition relation, where:*
   a) *G is a set of* guards*, i.e. boolean formulae over atoms of type $t \sim 0$, where t is a linear term over $Vars$ and $\sim \in \{\leq, =, \geq\}$;*
   b) *U is a set of* updates*, i.e. sets of assignments of type $v_j := f(v_0, \ldots, v_k)$, where $\forall_{0 \leq i \leq k} v_i \in Vars$,*

$v_j \in Vars$ *and* f *is a function whose domain and codomain are compatible with the domains of its arguments and target; it is assumed that each variable is assigned at most once per update;*

c) $Acts$ *is a finite set of action names;*

By a valuation of $Vars$ we mean a function $\omega\colon Vars \to \mathbb{N}$; the set of all valuations of $Vars$ is denoted by $Vals$. By $u(\omega) \in Vals$ we denote the valuation s.t. for $v_j \in Vars$:

$$u(\omega)(v_j) = \begin{cases} f(\omega(v_0), \ldots, \omega(v_k)) & \text{if } v_j := f(v_0, \ldots, v_k) \in u \\ \omega(v_j) & \text{otherwise} \end{cases}$$

By $g(\omega)$ we mean the boolean value of the expression obtained after valuating the variables in g with ω.

We write $s \xrightarrow[u]{g,act} s'$ instead of $(s, act, g, u, s') \in \to$. We also denote $acts(\mathcal{M}) = Acts$.

Definition 2 (Concrete Semantics of $\mathcal{GUS}$)**.** *Let* $\mathcal{M}$ *be a* $\mathcal{GUS}$ *and* $\omega^0 \in Vals$ *be an initial valuation of* $Vars$*. By the* concrete semantics *of* $\mathcal{M}$ *over* ω^0*, we mean a tuple* $\mathcal{CS}(\mathcal{M}, \omega^0) = \langle CS, w^0, \to \rangle$*, where:*

1) $w^0 = (s^0, \omega^0)$*;*
2) $CS = \mathcal{S} \times Vals$ *is the set of* concrete states*;*
3) $\to \subseteq CS \times Acts \times CS$ *is the transition relation s.t.* $(s, \omega) \xrightarrow{act} (s', \omega')$ *iff* $s \xrightarrow[u]{g,act} s'$ *where* $g(\omega)$ *is true and* $\omega' = u(\omega)$*, for some guard* g *and update* u*.*

A run $\rho = t_0 act_0 t_1 act_1 \ldots$ *is an infinite sequence of alternating concrete states and transitions s.t. for all* $i \in \mathbb{N}$ *we have* $t_i \xrightarrow{act_i} t_{i+1}$*. By* $Runs(\mathcal{M}, t)$ *we denote the set of all runs starting from* $t \in CS$*. We write* $Runs(\mathcal{M})$ *when the starting state is assumed to be initial.*

The systems tackled in this paper are composed of modules that can share variables and synchronise over common action labels. Hence we define the asynchronous product of $\mathcal{GUS}$.

Definition 3 (Asynchronous Product)**.** *Let* $\mathcal{M}_i = \langle \mathcal{S}_i, s_i^0, \to_i, Acts_i \rangle$ *be a* $\mathcal{GUS}$*, for* $i \in \{1, 2\}$*. The* asynchronous product *of* $\mathcal{M}_1$ *and* $\mathcal{M}_2$ *is the* $\mathcal{GUS}$ $\mathcal{M}_1 || \mathcal{M}_2 = \langle \mathcal{S}_1 \times \mathcal{S}_2, (s_1^0, s_2^0), \to, Acts_1 \cup Acts_2 \rangle$ *with the transition rule defined in the usual way:*

$$\frac{act \in Acts_1 \setminus Acts_2 \wedge s_1 \xrightarrow[u]{g,act}_1 s_1'}{(s_1, s_2) \xrightarrow[u]{g,act} (s_1', s_2)}$$

$$\frac{act \in Acts_2 \setminus Acts_1 \wedge s_2 \xrightarrow[u]{g,act}_2 s_2'}{(s_1, s_2) \xrightarrow[u]{g,act} (s_1, s_2')}$$

$$\frac{act \in Acts_1 \cap Acts_2 \wedge s_1 \xrightarrow[u_1]{g_1,act}_1 s_1' \wedge s_2 \xrightarrow[u_2]{g_2,act}_2 s_2'}{(s_1, s_2) \xrightarrow[u_1 \cup u_2]{g_1 \wedge g_2, act} (s_1', s_2')}$$

The last rule above can be applied only if $(u_1 \cup u_2)$ *is an update, i.e. each variable is assigned at most once.*

The above definition is naturally extended to an arbitrary number of components, where we sometimes write $||_{i=0}^n \mathcal{M}_i$ *instead of* $\mathcal{M}_1 || \ldots || \mathcal{M}_n$*.*

The synchronisation topology is a graph that records how components synchronise with one another.

Definition 4 (Synchronisation Topology)**.** *The* synchronisation topology *induced by a* $\mathcal{GUS}$ $\mathcal{G} = ||_{i=0}^n \mathcal{M}_i$ *is the undirected graph* $\mathcal{SG}(\mathcal{G}) = \langle \{\mathcal{M}_i \mid i = 0 \ldots n\}, \mathcal{E} \rangle$*, where* $(\mathcal{M}_i, \mathcal{M}_j) \in \mathcal{E}$ *iff* $i \neq j$ *and* $Acts_i \cap Acts_j \neq \emptyset$*.*

3.2. ADTree Semantics and Pattern-based Reduction

ADTrees to $\mathcal{GUS}$. The semantics of ADTrees used in this paper originates from [9]. Given an ADTree $\mathcal{T}$ with a set of nodes $\{A_i \mid i = 0 \ldots n\}$, a $\mathcal{GUS}$ M_i is associated with each A_i. The associated synchronisation topology $\mathcal{SG}(\mathcal{T})$ is defined by replacing each node A_i of $\mathcal{T}$ with the $\mathcal{GUS}$ M_i. Moreover, the attributes added to A_i are modelled by corresponding variables in $\mathcal{M}_i$ while the conditions added to A_i are represented by guards in $\mathcal{M}_i$. Variables are updated during the synchronisation between a child and its father node, mimicking how the values of the attributes are actually used. These variables may be used in the guards of the node's ancestors to check if the actual value that was set satisfies some condition. Note that the topology induced by an ADTree $\mathcal{T}$ is a tree. From now on, we focus on tree topologies.

In the figures, we employ message sender/recipient markings in a form of additional !/? action prefixes, which is only a syntactic sugar. It allows for an explicit synchronisation between two partners. In the particular case where there is no partner to synchronise with (root of the tree), this is a regular action.

Let us assume that the root of the tree $\mathcal{SG}(\mathcal{T})$ is labelled by $\mathcal{R}$. A run of $||_{i=0}^n \mathcal{M}_i$ describes a successful execution of $\mathcal{T}$ if it eventually stabilises on an infinite loop of executions of $\mathcal{R}\_ok$, i.e. the *good* final state is reachable. The runs that end with an infinite sequence of $\mathcal{R}\_nok$ are $\mathcal{T}$'s failures, i.e. the *bad* final state can be reached.

Pattern Reductions. In Table 2 we recall from [9] the basic reductions that can be used instead of the full patterns. These transformations, called *pattern reductions*, make the concrete semantics of the resulting synchronisation topology amenable, while keeping behaviours necessary to check ADTrees properties.

The key idea is that before pattern reduction a given node can receive all ok and nok messages from its children in any order, before it performs its own action and sends its messages. The reduction is simply a predefined selection of appropriate message orders and allows for constructing agent models according to the patterns shown in Table 2, while preserving the ADTree properties [9, Theorem 1].

Comparison with POR. This pattern-based reduction is in many aspects similar to partial order reduction (POR), a well-known and extensively studied technique for generating

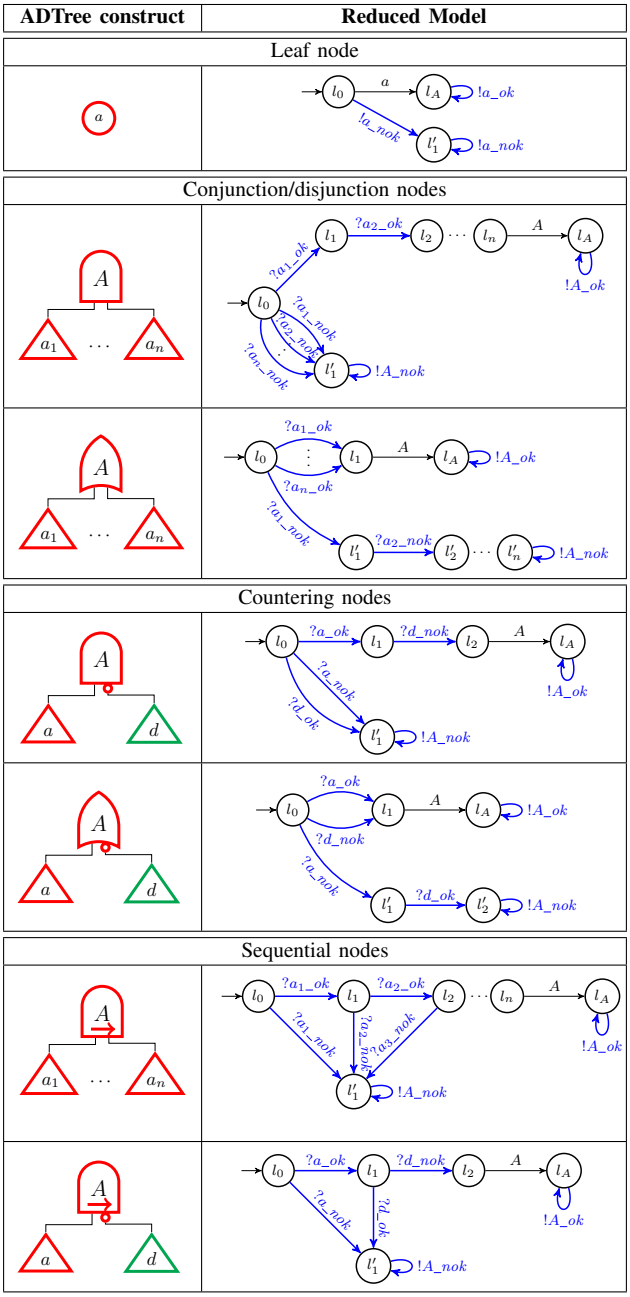


Table 2: Pattern reduction: ADTree nodes and $\mathcal{GUS}$ patterns

reduced state spaces in concurrent systems [19], [20], [21]. In particular, a single path of arbitrarily ordered transitions leads to the loop transition labelled with A_ok or A_nok for every node in the tree. This is precisely what one would expect, intuitively, from a POR algorithm: the actions of different child nodes are independent, so it does not matter in which order they are received. However, note that the pattern-based reduction goes further than that, truncating the path whenever possible in accordance with the nodes' semantics. For instance, AND nodes need not wait to hear from all children if one synchronises with a *nok* message.

While this already constitutes a slight improvement over the POR scheme, we can exploit the tree structure of $\mathcal{SG}(\mathcal{T})$ to obtain further (and independent!) *layered reductions*.

4. Layered Reductions

We now consider $\mathcal{GUS}$ $\mathcal{G} = ||_{i=0}^{n}\mathcal{M}_i$ with a tree synchronisation topology and show how, under certain assumptions, it can be exploited to obtain a reachability-preserving reduction of the system.

4.1. Tree Synchronisation Topologies

We now define the precedence relation between components of a synchronisation topology. Intuitively, a child precedes its father iff all the synchronisations between them occur before any other father action.

Definition 5 (Precedence)**.** *Let $\mathcal{M}_N, \mathcal{M}_C$ be a node and one of its children, respectively. If along each run $\rho \in Runs(\mathcal{G})$ after executing an action $act \in Acts_N \setminus Acts_C$ no action from $Acts_N \cap Acts_C$ appears, then we write $\mathcal{M}_C \hookrightarrow \mathcal{M}_N$ and say that $\mathcal{M}_C$* precedes *$\mathcal{M}_N$.*

Definition 6 (Root-directed Synchronisation Tree)**.** *A synchronisation tree $\mathcal{SG}(\mathcal{G})$ is* root-directed *if for each node $\mathcal{M}_N$ and any of its children $\mathcal{M}_C$ we have $\mathcal{M}_C \hookrightarrow \mathcal{M}_N$.*

To ensure that the runs in the product of a subtree do not depend on variable updates in disjoint subtrees we introduce the notion of update-separability.

Definition 7 (Update-separability)**.** *A root-directed synchronisation tree $\mathcal{SG}(\mathcal{G})$ is* update-separable *if for each $v \in Vars$ the following conditions hold:*

1) *v is updated in at most one component $\mathcal{M}_v$;*
2) *v is tested only in guards of the ancestors of node $\mathcal{M}_v$ in the tree $\mathcal{SG}(\mathcal{G})$.*

Example 2. *Figure 2 presents a simple tree synchronisation topology with components $\mathcal{M}_F$, its two children $\mathcal{M}_{N_1}$ and $\mathcal{M}_{N_2}$, the only child $\mathcal{M}_{C_1}$ of $\mathcal{M}_{N_1}$, and two children $\mathcal{M}_{C_2}, \mathcal{M}_{C_3}$ of $\mathcal{M}_{N_2}$. It can easily be checked that the tree is root-directed and is obviously update-separable as variable v is updated only at leaf $\mathcal{M}_{C_2}$ and read at the root $\mathcal{M}_F$.*

In what follows, we will manipulate subtrees of tree topologies.

Definition 8 (Subtree rooted in $\mathcal{M}_i$)**.** *The subtree of $\mathcal{SG}(\mathcal{G})$ rooted in $\mathcal{M}_i$, denoted by $\Downarrow\mathcal{M}_i$, is the tree containing $\mathcal{M}_i$ and all its descendants.*

Intuitively, the projection of a run ρ on a subtree $\Downarrow\mathcal{M}_i$, denoted by $\rho_{\Downarrow\mathcal{M}_i}$, is obtained by keeping from ρ only the locations, transitions and variables belonging to the nodes in $\Downarrow\mathcal{M}_i$.

Definition 9 (Projections on Subtrees)**.** *Let $\rho = t_0 act_0 t_1 act_1 \ldots$ be a run in $Runs(\mathcal{G})$ and $\mathcal{M}_i$ be a node of $\mathcal{SG}(\mathcal{G})$. The projection of ρ on $\Downarrow\mathcal{M}_i$, denoted by $\rho_{\Downarrow\mathcal{M}_i}$, is obtained by:*

1) *retaining in each concrete state* $t_j, j \in \mathbb{N}$ *only its projection (states and variables) on* $\Downarrow\mathcal{M}_i$*;*
2) *keeping only the transitions in the nodes of* $\Downarrow\mathcal{M}_i$*.*

Note that for any action not in the subtree, the projected source and target states are identical, and thus these actions are safely removed from the projected run.

Lemma 1. *Let* $\mathcal{SG}(\mathcal{G})$ *be a root-directed, update-separable tree. Let* $\mathcal{M}_i$ *be a node and* $\rho \in Runs(\mathcal{G})$*. Then,* $\rho_{\Downarrow\mathcal{M}_i}$ *is a prefix of some run* $\rho' \in Runs(\Downarrow\mathcal{M}_i)$*.*

Proof. (Sketch) It suffices to observe that by definition of update-separability the variables tested in $\rho_{\Downarrow\mathcal{M}_i}$ are updated only in $\Downarrow\mathcal{M}_i$. □

ADTrees are an example of such trees.

Lemma 2. *ADTrees topologies are root-directed and update-separable.*

Proof. The root-directed property follows from the patterns in Table 2. Indeed, all synchronisations with children occur before local actions and synchronisation with the father node. The update-separability stems from the variables being updated only when synchronising with children and tested by ancestors. □

We now outline the details of the layered reduction of tree topologies.

4.2. Layered Reduction for Trees

In the layered construction, we consider specifically the last synchronisation of node $\mathcal{M}_N$ with one of its children $\mathcal{M}_C$, before any other action in $\mathcal{M}_N$. Let $\#_{child}(\mathcal{M}_N)$ be the number of children of node $\mathcal{M}_N$.

Definition 10 (Last synchronisations with children)**.** *By the last synchronisations of* M_N *with its children we mean the transitions (denoted by lst), that are synchronising transitions of* M_N *and one of its children* $\mathcal{M}_{C_j}$*, such that there are states* s_i, s_{i+1}, s_{i+2} *and another transition* t *of* M_N *which does not synchronise with any transition of its children* $\mathcal{M}_{C_j}$*, with* s_i *lst* s_{i+1} t s_{i+2}*. The set of these transitions is denoted by* $Lst_C(\mathcal{M}_N)$*.*

Dealing with a single depth. Let us fix a depth $d > 0$ of the tree $\mathcal{SG}(\mathcal{G})$. We add a fresh variable v_d, initialised with 0, that counts the total number of synchronisations between the nodes at depth d and the nodes at depth $d+1$.

We modify each node $\mathcal{M}_N$ at depth d by adding to the update u of any transition in $Lst_C(\mathcal{M}_N)$ a new element $v_d := v_d + \#_{child}(\mathcal{M}_N)$. It is a way for node $\mathcal{M}_N$ to notify it has performed all synchronisations with its children.

The total number of the children of the nodes at depth d is $\#_{child}(d) = \#_{child}(\mathcal{M}_{N_1}) + \ldots + \#_{child}(\mathcal{M}_{N_k})$ where $\{\mathcal{M}_{N_1}, \ldots, \mathcal{M}_{N_k}\}$ are all the nodes at depth d in $\mathcal{SG}(\mathcal{G})$.

In the next step of the construction we also modify each node $\mathcal{M}_N$ at depth d by extending the guard g of each transition t of M_N which does not synchronise with any transition of the children of M_N to $g \wedge (v_d = \#_{child}(d))$.

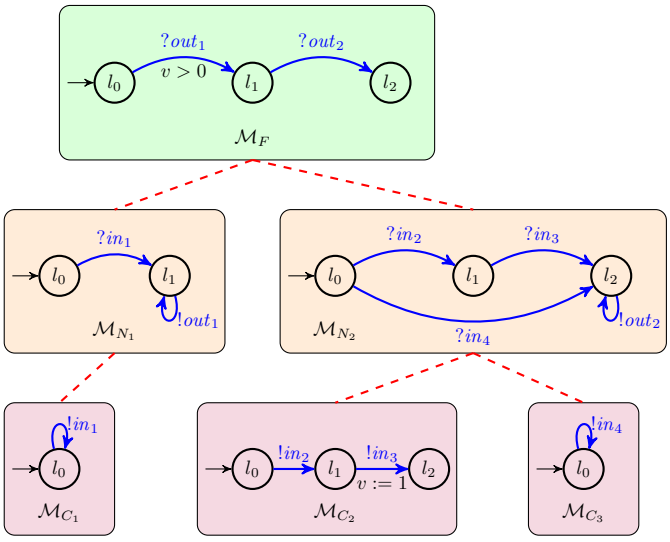


Figure 2: Example of a $\mathcal{GUS}$ with a tree synchronisation topology

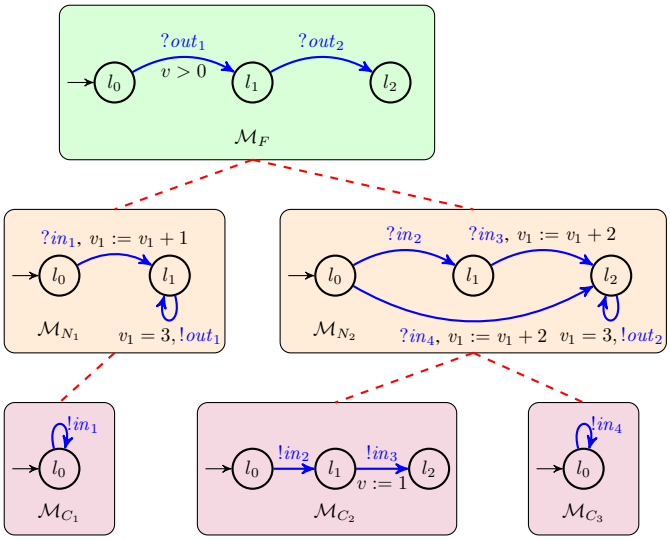


Figure 3: Example of layered reduction

This, intuitively, prevents any action at depth d before all synchronisations with the children are finished.

Dealing with the entire tree. In order to obtain the final result, denoted by $\mathcal{SG}^{lr}(\mathcal{G})$, the above transformation is performed for each depth $0 < d <$ height of $\mathcal{SG}(\mathcal{G})$.

Example 3. *Let us revisit Example 2 with the layered reduction scheme. The resulting tree is shown in Figure 3. To this end we add a new variable* v_1 *used at depth 1 in the tree. Note that in the transformed model nodes* $\mathcal{M}_1$ *and* $\mathcal{M}_2$ *must wait until they both fully synchronise with their children before they can synchronise with* $\mathcal{M}_F$*.*

The following proposition states that the technique of

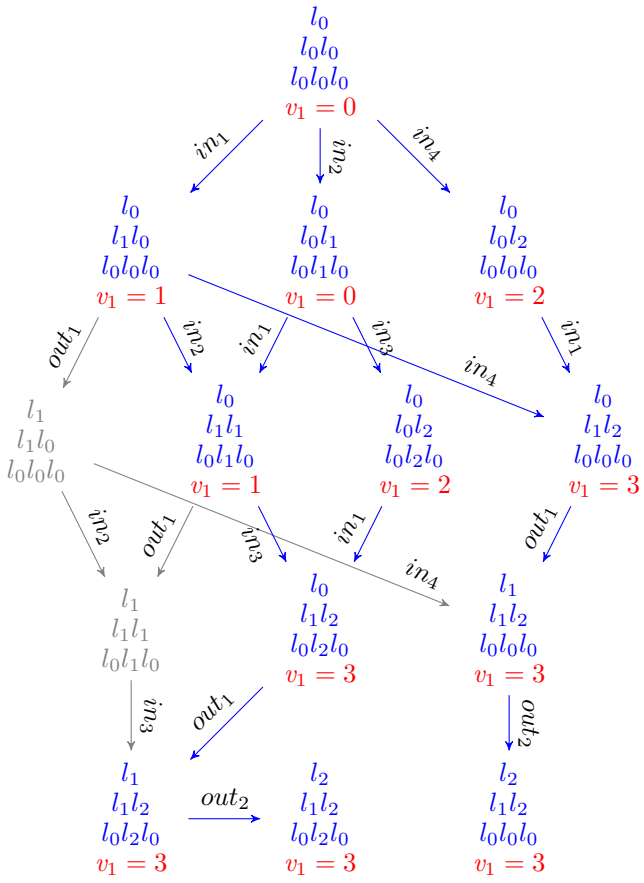


Figure 4: State space for the $\mathcal{GUS}$ from Figure 3: full (14 states, 19 edges) vs. layered reduction (12 states, 14 edges)

layered reductions preserves reachability.

Proposition 1. *Let $\mathcal{SG}(\mathcal{G})$ be a root-directed, update-separable tree. A good (bad) final state s is reachable in $\mathcal{SG}(\mathcal{G})$ iff it is reachable in $\mathcal{SG}^{lr}(\mathcal{G})$.*

Proof. (Sketch) It follows from Lemma 1 and the construction of $\mathcal{SG}^{lr}(\mathcal{G})$. □

Example 4. *Figure 4 shows the state space of the $\mathcal{GUS}$ from Figure 3. The grey nodes and edges appear in the full state space but not in the layered reduction.*

Let us move to the experimental evaluation of our approach.

5. Experiments

In the paper [9] we showed the benefits of our approach against other existing works as far computing minimum cost and maximum time is concerned. Here, we evaluate the effectiveness of our reductions with IMITATOR (www.imitator.fr, [10]), a model-checker for Parametric Timed Automata [22], where models are composed of automata that synchronise on shared transitions. The choice of the IMITATOR tool is motivated by: the easy encoding of our models, including discrete variables (for costs and time in the ADTrees); the capability to output the state space for exploration purposes; the possible use of parameters in place of some variables, for synthesizing adequate values. Experiments were executed on a 2.7 GHz Intel Core i7, with 16 GB of memory, using a timeout of 30 minutes. All files used in the experiments are available at http://lipn.univ-paris13.fr/~petrucci/ICECCS19_imitator_files.zip.

5.1. ADTree Case Studies

The first set of experiments concerned the treasure hunters case study (see Example 1), as well as three more from [9], which we briefly recall below.

Forestalling a software release (forestall)**.** Based on a real-world instance [4], this case study involves an unlawful competitor attempting to steal the intellectual property of a company so as to be the first to release a software product, gaining a market advantage.

Compromise IoT device (iot-dev)**.** A LAN-based attack against an Internet-of-Things device, where the attacker, having obtained credentials for the private network, exploits a software vulnerability to execute a malicious script. The original attack tree [23], later extended in [7], is further enriched by adding defense mechanisms.

Obtain admin privileges (gain-admin)**.** An attempt to gain administrative permissions on a UNIX system, either by physically accessing an already logged-in console or remotely via privilege escalation. The original attack tree from [24] is extended with the SAND gate [25] and reactive defences.

The results are summarised in Table 3. For each model, we report the number of states, the number of transitions in the state space and its generation time in four cases:

1) the **full** model for each ADTree node, that comprises all possible combinations of messages received from children as in [9] prior to pattern reduction,
2) the model reduced with the **layers** reduction scheme introduced in Section 4,
3) the model reduced to the **patterns** defined in [9] and recalled in Table 2,
4) the model obtained with **both** of these reductions (patterns and layers);
5) the size of the version with both reductions compared with the others (highlighted in green).

All experiments show that applying both reduction types together (patterns and layers) is the most efficient, and when applied individually, the former performs better.

Moreover, in line with the theory in Section 4, we have confirmed that in all cases exploring the state space gives the minimum and maximum cost and time for an attack to be successful. The gain-admin example timed out except when using both reductions.

Model	No reduction			Layers			Patterns			Both			% size		
	$\|S\|$	$\|T\|$	t (s)	$\|S\|$	$\|T\|$	t (s)	$\|S\|$	$\|T\|$	t (s)	$\|S\|$	$\|T\|$	t (s)	$\frac{\|S\|_{Both}}{\|S\|_{No}}$	$\frac{\|S\|_{Both}}{\|S\|_{Layer}}$	$\frac{\|S\|_{Both}}{\|S\|_{Pattern}}$
treasure hunters	558	1,452	0.322	278	452	0.129	156	339	0.082	74	89	0.041	13.26%	26.62%	47.44%
forestall	77,803	215,349	542.976	39,893	67,862	150.775	7,390	22,250	39.390	1,845	2,721	4.094	2.37%	4.62%	24.97%
iot-dev	4,349	8,280	11.280	3,145	4,050	6.510	907	2,154	2.207	371	450	0.623	8.53%	11.80%	40.90%
gain-admin	TO			TO			TO			52,923	94,570	416.431			

Table 3: Experimenting the different reductions on the case studies from [9]

Furthermore, although Table 3 shows the results for fixed initial values of cost and time, the parametric versions of the case studies were also successful in synthesizing parameter values, e.g. for the robbery to fail in the treasure hunters example, the police should arrive in less than 5 minutes (the police time was then a parameter).

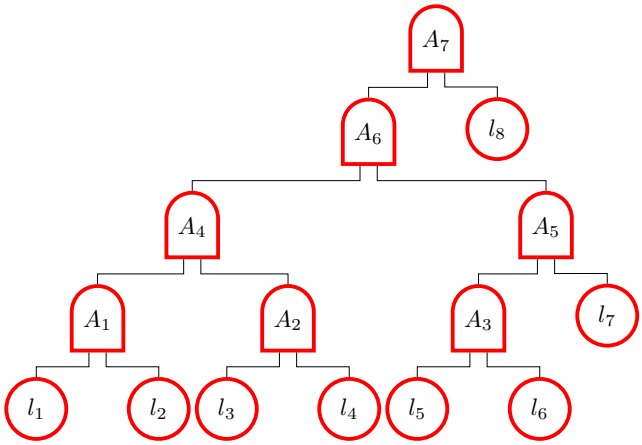


Figure 5: ADTree generated for 2 children per node, of depth 4 and width 6

5.2. Scalable Experiments

To obtain a better insight into the gains entailed by each reduction, we conducted a second set of experiments. For that purpose, we wrote a generator of ADTrees that outputs the four IMITATOR files according to which reductions are used. These ADTrees allowed for scalable experiments, as reported in Table 4, and pictured in Figures 6 and 7.

The left part of Table 4, in grey, shows the parameters of the automatic generation:

1) a fixed *number of children* for each intermediate node in the ADTree[1];
2) the number of nodes in the ADTree, indicated for information, as this is entailed by the other numbers;
3) the *depth* of the generated ADTree;
4) a *width* corresponding to the number of deepmost nodes; nodes at some depth are used as children of nodes at the previous depth, and leaves may be added for the parent to have the specified number of children (see example in Figure 5).

The sizes of the state spaces are then collected in the table, and the size of the version with both reductions compared with the others in the rightmost columns (highlighted in green). The results show that the behaviour is similar to that of the previous case studies. We also note that the gain of reductions can be huge, as e.g. for 3 children, depth 3 and width 9, the reduction leads to a state space that is 0.98% the size of the full one (line 27 of Table 4). For the same number of children and the same width, the varying depth has little impact on the reduced state space: although there are a few more nodes, the interleavings are limited. This can be observed for e.g. 2 children and width 6, at lines 3, 6, 10, 14 and 18. On the contrary, for the same number of children and the same depth, the width has a huge impact: limiting the interleavings with the layered reduction shows its efficiency. This can be seen for e.g. 2 children and depth 3, at lines 2–4.

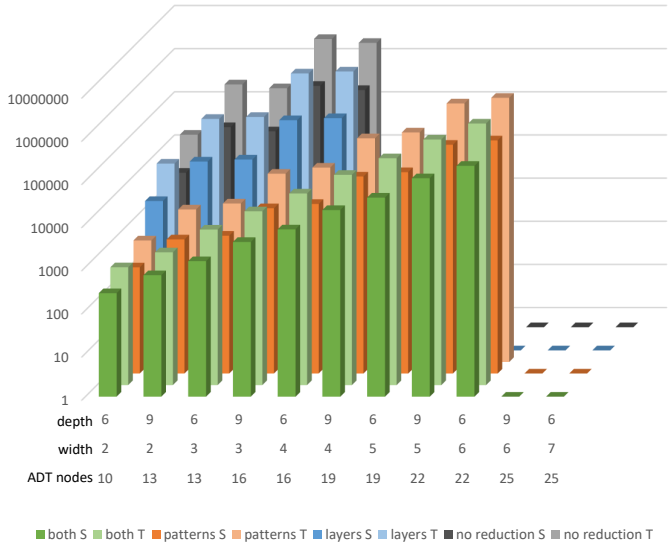


Figure 6: Scalability of different reductions for ADTrees with 3 children (missing bars indicate timeouts)

6. Conclusion

Attack-defence trees are a graphical model to represent systems and analyse the efficiency of countermeasures against possible attacks, and determine the necessary conditions for attacks to take place. These systems include some

[1]The intermediate nodes generated are all of the `AND` or the `OR` type. These two cases lead to exactly the same results as they exhibit an identical automaton structure. Only synchronisation labels are inverted.

Model				No reduction		Layers		Patterns		Both		% size			
#child.	$\|ADTree\|$	depth	width	$\|S\|$	$\|T\|$	$\|S\|$	$\|T\|$	$\|S\|$	$\|T\|$	$\|S\|$	$\|T\|$	$\frac{\|S\|_{Both}}{\|S\|_{No}}$	$\frac{\|S\|_{Both}}{\|S\|_{Layer}}$	$\frac{\|S\|_{Both}}{\|S\|_{Pattern}}$	
2	7	2	4	185	432	145	296	72	123	54	69	29.19%	37.24%	75.00%	1
2	9	3	4	587	1,698	467	1,210	246	571	190	373	32.36%	40.68%	77.23%	2
2	13	3	6	8,823	35,602	4,043	14,046	2,405	7,584	883	1,802	10.00%	21.84%	36.71%	3
2	15	3	8	34,481	160,096	11,425	44,464	6,734	23,135	1,808	3,439	5.24%	15.82%	26.84%	4
2	11	4	4	1,825	6,332	1,465	4,628	840	2,458	652	1,680	35.72%	44.50%	77.62%	5
2	15	4	6	26,725	124,708	12,385	50,480	8,184	30,773	3,256	9,167	12.18%	26.28%	39.78%	6
2	17	4	8	103,955	549,762	34,787	156,754	22,854	92,393	6,606	17,615	6.35%	18.99%	28.90%	7
2	23	4	10	TO		TO		TO		71,965	237,154				8
2	13	5	4	5,603	22,774	4,523	16,942	2,868	10,124	2,228	7,088	39.76%	49.26%	77.68%	9
2	17	5	6	80,687	428,086	37,667	176,722	27,926	121,887	11,258	38,693	13.95%	29.89%	40.31%	10
2	19	5	8	312,889	1,858,220	105,385	540,860	77,948	362,276	22,808	74,986	7.29%	21.64%	29.26%	11
2	25	5	10	TO		TO		TO		273,484	1,128,571				12
2	15	6	4	17,065	79,784	13,825	60,128	9,792	40,476	7,608	28,796	44.58%	55.03%	77.69%	13
2	19	6	6	243,085	1,446,656	114,025	606,524	95,336	473,670	38,520	155,698	15.84%	33.78%	40.40%	14
2	21	6	8	TO		318,203	1,835,398	266,084	1,397,360	78,020	303,724		24,51%	29.32%	15
2	27	6	10	TO		TO		TO		TO					16
2	17	7	4	51,707	273,994	41,987	208,546	33,432	158,372	25,976	113,996	50.23%	61.86%	77.69%	17
2	21	7	6	TO		344,123	2,049,670	325,492	1,813,652	131,564	611,188		38.23%	40.42%	18
2	23	7	8	TO		TO		TO		266,464	1,198,156				19
2	19	8	4	156,145	926,420	126,985	710,636	114,144	609,608	88,688	442,736	56.79%	69.84%	77.69%	20
2	23	8	6	TO		TO		TO		TO					21
2	21	9	4	TO		TO		TO		302,800	1,694,352				22
2	23	10	4	TO		TO		TO		TO					23
3	10	2	6	3,803	15,598	2,891	11,486	289	654	250	537	6.57%	8.64%	86.50%	24
3	13	2	9	43,387	228,362	23,779	124,754	1,283	3,424	653	1,192	1.50%	2.74%	50.89%	25
3	13	3	6	34,739	186,530	26,531	138,578	1,563	4,700	1,380	4,025	3.97%	5.20%	88.29%	26
3	16	3	9	392,531	2,577,950	216,059	1,410,182	6,771	23,024	3,846	10,667	0.98%	1.78%	56.80%	27
3	16	4	6	314,699	2,097,686	240,827	1,567,622	8,511	31,912	7,530	27,559	2.39%	3.12%	88.47%	28
3	19	4	9	TO		TO		36,777	152,390	21,117	74,504			57.41%	29
3	19	5	6	TO		TO		46,377	208,290	41,040	180,639			88.49%	30
3	22	5	9	TO		TO		200,349	978,834	115,164	491,799			57.48%	31
3	22	6	6	TO		TO		252,729	1,322,490	223,650	1,150,257			88.49%	32
3	25	6	9	TO		TO		TO		TO					33
3	25	7	6	TO		TO		TO		TO					34

Table 4: Scalability of the different reductions

attributes such as the cost of an attack/defence or the time it takes. Although previous works have proposed automata-based models of ADTrees, the state space explosion limits the formal analysis capabilities. In this paper, we have shown the soundness of our former translation to reduced patterns.

Moreover, we defined Guarded Update Systems, which are automata manipulating variables and proposed to exploit the characteristics of those that exhibit a tree structure to further contain state space explosion by avoiding some unnecessary interleavings. This approach directly applies to ADTrees, in combination with the previous one, and extensive experiments proved its efficiency. We believe that the approach applies not only to tree topologies but also to directed acyclic graphs (DAGs), and thus to models from other application domains such as workflows.

This layered reduction opens the perspective of compositional analysis where subtrees could be handled and their output inputed to the analysis of their parent tree. It would then also be possible to achieve parallel model-checking of systems with a tree topology in general and ADTrees in particular.

Another perspective is to extend these state space reductions to models that take into account the assignment of agents to ADTree nodes, as in [9], so as to analyse possible coalitions in attacks and/or defences.

References

[1] B. Kordy, S. Mauw, S. Radomirović, and P. Schweitzer, "Foundations of attack-defense trees," in *FAST 2010*, ser. LNCS, vol. 6561. Springer, 2011, pp. 80–95.

[2] Z. Aslanyan and F. Nielson, "Pareto Efficient Solutions of Attack-Defence Trees," in *Principles of Security and Trust*. Springer Berlin Heidelberg, 2015, vol. 9036, pp. 95–114.

[3] S. Mauw and M. Oostdijk, "Foundations of Attack Trees," in *ICISC 2005*. Springer, 2006, pp. 186–198.

[4] A. Buldas, P. Laud, J. Priisalu, M. Saarepera, and J. Willemson, "Rational Choice of Security Measures Via Multi-parameter Attack Trees," in *Critical Information Infrastructures Security*. Springer, 2006, pp. 235–248.

[5] R. Jhawar, B. Kordy, S. Mauw, S. Radomirović, and R. Trujillo-Rasua, "Attack trees with sequential conjunction," in *ICT Systems Security and Privacy Protection*. Springer, 2015, pp. 339–353.

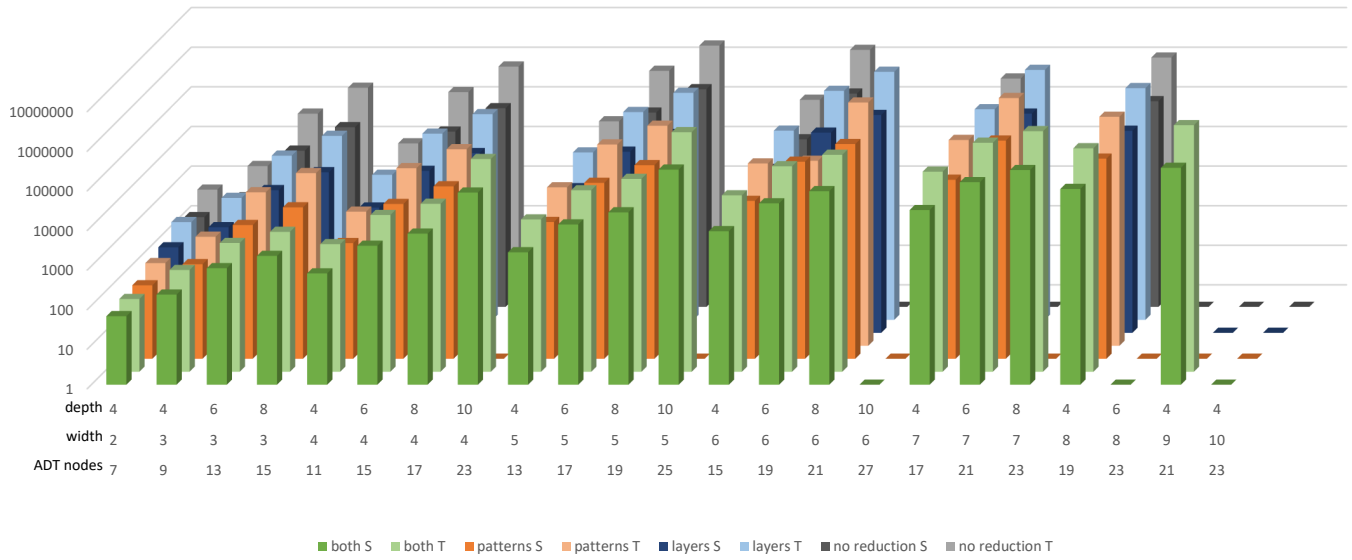


Figure 7: Scalability of different reductions for ADTrees with 2 children (missing bars indicate timeouts)

[6] B. Kordy, L. Piètre-Cambacédès, and P. Schweitzer, "DAG-based attack and defense modeling: Don't miss the forest for the attack trees," *Computer Science Review*, vol. 13-14, pp. 1–38, 2014.

[7] R. Kumar, S. Schivo, E. Ruijters, B. M. Yildiz, D. Huistra, J. Brandt, A. Rensink, and M. Stoelinga, "Effective analysis of attack trees: A model-driven approach," in *Fundamental Approaches to Software Engineering*. Springer, 2018, pp. 56–73.

[8] C. Salter, O. Saydjari, B. Schneier, and J. Wallner, "Toward a secure system engineering methodolgy," in *NSPW'98*. ACM, 1998, pp. 2–10.

[9] J. Arias, C. Budde, W. Penczek, L. Petrucci, and M. Stoelinga, "Hackers vs. Security: Attack-Defence Trees as Asynchronous Multi-Agent Systems," 2019, https://arxiv.org/abs/1906.05283.

[10] É. André, L. Fribourg, U. Kühne, and R. Soulat, "IMITATOR 2.5: A tool for analyzing robustness in scheduling problems," in *FM 2012*, ser. LNCS, vol. 7436. Springer, 2012, pp. 33–36.

[11] B. Kordy, S. Mauw, S. Radomirović, and P. Schweitzer, "Attack–defense trees," *Journal of Logic and Computation*, vol. 24, no. 1, pp. 55–87, 2014.

[12] O. Gadyatskaya, R. R. Hansen, K. G. Larsen, A. Legay, M. C. Olesen, and D. B. Poulsen, "Modelling Attack-defense Trees Using Timed Automata," in *Formal Modeling and Analysis of Timed Systems*, M. Fränzle and N. Markey, Eds. Springer International Publishing, 2016, vol. 9884, pp. 35–50.

[13] F. Arnold, D. Guck, R. Kumar, and M. Stoelinga, "Sequential and parallel attack tree modelling," in *Computer Safety, Reliability, and Security*. Springer International Publishing, 2015, pp. 291–299.

[14] M. Gribaudo, M. Iacono, and S. Marrone, "Exploiting bayesian networks for the analysis of combined attack trees," *Electronic Notes in Theoretical Computer Science*, vol. 310, pp. 91–111, 2015.

[15] Z. Aslanyan, F. Nielson, and D. Parker, "Quantitative Verification and Synthesis of Attack-Defence Scenarios," in *CSF 2016*. IEEE, 2016, pp. 105–119.

[16] É. André, D. Lime, M. Ramparison, and M. Stoelinga, "Parametric analyses of attack-fault trees," in *Proc. 19th Int. Conf. on Application of Concurrency to System Design (ACSD'19), Aachen, Germany*. IEEE, Jun. 2019, to appear.

[17] H. Hermanns, J. Krämer, J. Krčál, and M. Stoelinga, "The Value of Attack-Defence Diagrams," in *POST 2016*, ser. LNCS, vol. 9635. Springer, 2016, pp. 163–185.

[18] S. A. Çamtepe and B. Yener, "Modeling and detection of complex attacks," in *SecureComm*, 2007, pp. 234–243.

[19] P. Godefroid, *Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem*, J. v. Leeuwen, J. Hartmanis, and G. Goos, Eds. Berlin, Heidelberg: Springer-Verlag, 1996.

[20] D. Peled, "All from one, one for all: on model checking using representatives," in *Computer Aided Verification*, C. Courcoubetis, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 1993, pp. 409–423.

[21] A. Valmari, "Stubborn sets for reduced state space generation," in *Advances in Petri Nets 1990*, G. Rozenberg, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 1991, pp. 491–515.

[22] R. Alur, T. A. Henzinger, and M. Y. Vardi, "Parametric real-time reasoning," in *ACM Symposium on Theory of Computing, 1993*. ACM, 1993, pp. 592–601.

[23] M. Steiner and P. Liggesmeyer, "Qualitative and quantitative analysis of CFTs taking security causes into account," in *Computer Safety, Reliability, and Security*. Springer, 2015, pp. 109–120.

[24] J. D. Weiss, "A system security engineering process," in *Proceedings of the 14th National Computer Security Conference*, 1991, pp. 572–581.

[25] R. Kumar, E. Ruijters, and M. Stoelinga, "Quantitative attack tree analysis via priced timed automata," in *FORMATS 2015*, ser. LNCS, vol. 9268. Springer, 2015, pp. 156–171.