Modular Analysis of Tree-Topology Models

Jaime Arias, Michal Knapik, Wojciech Penczek, Laure Petrucci

Modular Analysis of Tree-Topology Models*

Jaime Arias[1][0000−0003−3019−4902], Michał Knapik[2][0000−0003−3259−9786], Wojciech Penczek[2][0000−0001−6477−4863], and Laure Petrucci[1][0000−0003−3154−5268]

[1] LIPN, CNRS UMR 7030, Université Sorbonne Paris Nord, 99 av. J-B. Clément, 93430 Villetaneuse, France {first.last}@lipn.univ-paris13.fr
[2] Institute of Computer Science, Polish Academy of Sciences, Jana Kazimierza 5, 01-248 Warsaw, Poland {first.last}@ipipan.waw.pl

Abstract. Networks of automata that synchronise over shared actions are organised according to a graph synchronisation topology. In this topology two automata are connected if they can jointly execute some action. We present a very effective reduction for networks with tree-like synchronisation topologies such that all automata after synchronising with their parents can execute only local (non-synchronising) actions: forever or until *resetting*, i.e. entering the initial state. We show that the reduction preserves reachability, but not liveness. This construction is extended to tree-like topologies of arbitrary automata and investigated experimentally.

Keywords: Model Checking · Networks of Synchronising Automata· State Space Reduction · Tree-Like Synchronisation Topologies.

1 Introduction

Networks of various types of finite automata (or labelled transition systems, LTSs for short) [10,14] are a popular choice of formalism when modelling complex systems such as protocols. In this approach, the components of the system under investigation are abstracted as state machines and the behaviour of the entire system is captured by their synchronised product. However, the cost of computing the synchronised product can be prohibitive: in practice the size of the state space grows exponentially with the number of the sub-modules. This observation led to an extensive research into how to analyse and represent models that consist of interacting components without building the entire state space.

In this paper, we tackle the problem of computing a compact representation of the state space of the entire product of networks of LTSs that exhibit tree-like synchronisation topologies. Intuitively, this means a graph, whose edges depict synchronisation between LTSs over common actions of a system, is a

* The authors acknowledge the support of CNRS and PAN, under the IEA project MoSART, and of NCBR Poland and FNR Luxembourg, under the PolLux/FNR-CORE project STV (POLLUX-VII/1/2019).

tree. Examples of systems having such a synchronisation topology are e.g. attack-defense trees (ADT) [17,7,6,9], hierarchical [4], broadcast [12], multimedia [8], and workflow models [3].

This structure of communication is quite natural also in the system design where circular dependencies are sometimes treated as anti-patterns and cause problems such as deadlocks.

We design algorithms that compute a compact representation of the synchronised product bottom-up, focusing at each level only on the synchronisations between the pairs consisting of a parent and one of its children. As it turns out, the notion of memory is crucial to the size of the constructed LTS (called here the *sum-of-squares product*). Namely, if every component resets (i.e. returns to its initial state) after synchronising with its parent or enters a deadlock, then no information about its post-synchronisation states needs to be preserved. In this case, the sum-of-squares product is quite small and can be computed efficiently. In case some LTSs do not reset after synchronising with their parents, an additional memory gadget is needed. Then, the general sum-of-squares product also preserves reachability, but a reduction or good performance is not guaranteed. The burden of recording the post-synchronisation state of the components can even lead to substantial blowup of the state space.

Outline of the Paper

In Section 2 we provide the basic definitions of synchronising LTSs, live-reset and sync-deadlock LTSs, and synchronisation topologies. A bottom-up reduction for the networks of live-reset and sync-deadlock LTSs is presented in Section 3. We show how to transform these networks in a reachability-preserving way into a new, smaller model called the sum-of-squares product. The key idea is that the new model traces the interactions between the components and their parents, followed by upstream synchronisations. This bottom-up reduction is extended in Section 4 to a wider case of tree-like synchronisation topologies with general LTSs. Section 5 investigates experimentally the effectiveness of the reductions on several scalable examples and many random cases using a novel open-source tool [2]. Section 6 draws conclusions and discusses future work.

2 Tree Synchronisation Systems

In this section we recall the notions of networks of Labelled Transition Systems and their synchronisation topologies. We also introduce and explain the restrictions on the models assumed in the next section. In what follows $\mathcal{PV}$ denotes the set of propositions.

Definition 1 (Labelled Transition System). *A* Labelled Transition System *($\mathcal{LTS}$) is a tuple $\mathcal{M} = \langle \mathcal{S}, s^I, Acts, \rightarrow, \mathcal{L} \rangle$ where:*

1. *$\mathcal{S}$ is a finite set of states and $s^I \in \mathcal{S}$ the initial state;*

2. *Acts is a finite set of action names;*
3. $\rightarrow \subseteq \mathcal{S} \times Acts \times \mathcal{S}$ *is a transition relation;*
4. $\mathcal{L}\colon \mathcal{S} \to 2^{\mathcal{PV}}$ *is a labelling function.*

We usually write $s \xrightarrow{act} s'$ instead of $(s, act, s') \in \rightarrow$. We also denote $acts(\mathcal{M}) = Acts$ and $states(\mathcal{M}) = \mathcal{S}$. A *run* in $\mathcal{LTS}$ $\mathcal{M}$ is an infinite sequence of states and actions $\rho = s^0 act^0 s^1 act^1 \ldots$ s.t. $s^i \xrightarrow{act^i} s^{i+1}$ for all $i \geq 0$. By $Runs(\mathcal{M}, s)$ we denote the set of all the runs starting from state $s \in \mathcal{S}$. If s is the initial state, we write $Runs(\mathcal{M})$ instead of $Runs(\mathcal{M}, s^I)$.

2.1 $\mathcal{LTS}$ Networks and Synchronisation Topologies

Model checkers such as SPIN [13], UPPAAL [11], and IMITATOR [5] usually expect the systems to be described in a form of interacting modules. Synchronisation over common actions [10] (or *channels*) is a popular primitive that enables such an interaction.

Definition 2 (Asynchronous Product). *Let* $\mathcal{M}_i = \langle \mathcal{S}_i, s_i^I, Acts_i, \rightarrow_i, \mathcal{L}_i \rangle$ *be* $\mathcal{LTS}$*, for* $i \in \{1, 2\}$*. The* asynchronous product *of* $\mathcal{M}_1$ *and* $\mathcal{M}_2$ *is the* $\mathcal{LTS}$ $\mathcal{M}_1 || \mathcal{M}_2 = \langle \mathcal{S}_1 \times \mathcal{S}_2, (s_1^I, s_2^I), Acts_1 \cup Acts_2, \rightarrow, \mathcal{L}_{1,2} \rangle$ *s.t.* $\mathcal{L}_{1,2}((s_1, s_2)) = \mathcal{L}_1(s_1) \cup \mathcal{L}_2(s_2)$ *for all* $(s_1, s_2) \in \mathcal{S}_1 \times \mathcal{S}_2$ *and the transition rule is defined as follows:*

$$\frac{act \in Acts_1 \setminus Acts_2 \wedge s_1 \xrightarrow{act}_1 s_1'}{(s_1, s_2) \xrightarrow{act} (s_1', s_2)} \qquad \frac{act \in Acts_2 \setminus Acts_1 \wedge s_2 \xrightarrow{act}_2 s_2'}{(s_1, s_2) \xrightarrow{act} (s_1, s_2')}$$

$$\frac{act \in Acts_1 \cap Acts_2 \wedge s_1 \xrightarrow{act}_1 s_1' \wedge s_2 \xrightarrow{act}_2 s_2'}{(s_1, s_2) \xrightarrow{act} (s_1', s_2')}$$

The above definition is naturally extended to an arbitrary number of components. We sometimes write $||_{i=1}^n \mathcal{M}_i$ instead of $\mathcal{M}_1 || \ldots || \mathcal{M}_n$. If s is a state of $\mathcal{M}_1 || \ldots || \mathcal{M}_n$, then by $s_{\mathcal{M}_i}$ we denote its component corresponding to $\mathcal{M}_i$.

The synchronisation topology [17] is an undirected graph that records how LTSs synchronise with one another.

Definition 3 (Synchronisation Topology). *A* synchronisation topology *($\mathcal{ST}$) is a tuple* $\mathcal{G} = \langle Net, \mathcal{T} \rangle$*, where* $Net = \{\mathcal{M}_i\}_{i=1}^n$ *is a set of* $\mathcal{LTS}$*s for* $1 \leq i \leq n$*, and* $\mathcal{T} \subseteq Net \times Net$ *is s.t.* $(\mathcal{M}_i, \mathcal{M}_j) \in \mathcal{T}$ *iff* $i \neq j$ *and* $Acts_i \cap Acts_j \neq \emptyset$*.*

Note that $\mathcal{T}$ is induced by Net. Thus, with a slight notational abuse we sometimes treat $\mathcal{G}$ as Net. Moreover, we write $acts(\mathcal{G}) = \bigcup_{i=1}^n acts(\mathcal{M}_i)$.

Definition 4 (Tree Synchronisation Topology). *A synchronisation topology* $\mathcal{G}$ *s.t.* $\mathcal{T}$ *is a tree rooted in* $root(\mathcal{G})$ *is called* tree synchronisation topology*.*

It should be noted that $root(\mathcal{G})$ is not always uniquely induced by $\mathcal{G}$, the root is thus a part of the signature of tree synchronisation topology.

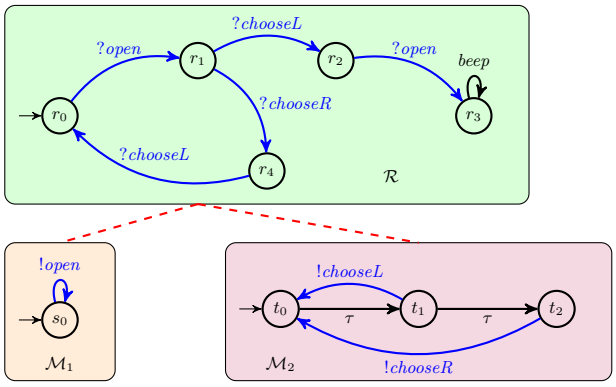

Fig. 1. A simple tree synchronisation topology $\mathcal{G}_x$.

Let us fix a tree synchronisation topology $\mathcal{G} = \langle Net, \mathcal{T}, root(\mathcal{G})\rangle$. For each $\mathcal{M} \in Net$ by $parent(\mathcal{M})$ we denote the parent of $\mathcal{M}$ in $\mathcal{T}$ (we assume that $parent(root(\mathcal{G})) = \emptyset$). By $children(\mathcal{M})$ we mean the set of the children of $\mathcal{M}$. By $upacts(\mathcal{M})$ (resp. $downacts(\mathcal{M})$) we denote the set of actions via which $\mathcal{M}$ synchronises with its parent (resp. children). For each $act \in downacts(\mathcal{M})$, $snd(\mathcal{M}, act)$ denotes the component $\mathcal{M}' \in children(\mathcal{M})$ s.t. $act \in upacts(\mathcal{M}')$. Thus, $snd(\mathcal{M}, act)$ is the child of $\mathcal{M}$ that synchronises with $\mathcal{M}$ over act. If $\mathcal{M}$ is clear from the context, we simply write $snd(act)$. The *local* unsynchronised actions of $\mathcal{M}$ are defined as $locacts(\mathcal{M}) = acts(\mathcal{M}) \setminus (downacts(\mathcal{M}) \cup upacts(\mathcal{M}))$.

For brevity, whenever we refer to a state, action label, transition, or state space of $\mathcal{G}$ we mean a state, action label, transition, or state space of $||_{i=1}^{n}\mathcal{M}_i$. We also extend the notion of runs to synchronisation topologies: $Runs(\mathcal{G}, s) = Runs(||_{i=1}^{n}\mathcal{M}_i, s)$ for each $s \in states(||_{i=1}^{n}\mathcal{M}_i)$. Moreover, for each $F \subseteq acts(\mathcal{G})$ we introduce the notion of *one-shot* F runs, $Runs_F(\mathcal{G}, s)$, as the set of all the runs in $Runs(\mathcal{G}, s)$ along which each action from F appears at most once.

Example 1. Figure 1 presents a small tree $\mathcal{ST}$ $\mathcal{G}_x$ with the root $\mathcal{R}$ and two children $\mathcal{M}_1$ and $\mathcal{M}_2$. The auxiliary symbols !/? are syntactic sugar, used to distinguish between *upacts* and *downacts*. Here, $upacts(\mathcal{R}) = \emptyset$, $downacts(\mathcal{R}) = \{open, chooseL, chooseR\}$, and $locacts(\mathcal{R}) = \{beep\}$. Similarly, $upacts(\mathcal{M}_1) = \{open\}$, $upacts(\mathcal{M}_2) = \{chooseL, chooseR\}$, $downacts(\mathcal{M}_1) = locacts(\mathcal{M}_1) = downacts(\mathcal{M}_2) = \emptyset$, and $locacts(\mathcal{M}_2) = \tau$.

We start by dealing with networks whose components all share a similar, simple structure.

Definition 5 (Live-Reset LTS). *A LTS $\mathcal{M}$ with initial state $s^I_{\mathcal{M}}$ is* live-reset *if for each run $\rho = s^0 act^0 s^1 act^1 \ldots$: $\forall i \in \mathbb{N}$ if $act^i \in upacts(\mathcal{M})$, then $s^{i+1} = s^I_{\mathcal{M}}$.*

Definition 6 (Sync-Deadlock LTS). *A LTS $\mathcal{M}$ is* sync-deadlock *if for each run $\rho = s^0 act^0 s^1 act^1 \ldots$: $\forall i \in \mathbb{N}$ if $act^i \in upacts(\mathcal{M})$, then for each $j > i$, $act^j \in locacts(\mathcal{M})$.*

LTSs that are either live-reset or sync-deadlock are said to be *sync-memoryless*[3]. Intuitively, after synchronising with its parent a sync-memoryless LTS either immediately enters its initial state (if live-reset) or executes only local actions (if sync-deadlock).

If every LTS of $\mathcal{ST}$ $\mathcal{G}$ is live-reset (resp. sync-deadlock), then we say that $\mathcal{G}$ is live-reset (resp. sync-deadlock). It is easy to see that the tree $\mathcal{ST}$ in Figure 1 is live-reset.

Definition 7 (Reachability and liveness). *For each $p \in \mathcal{PV}$ we write $\mathcal{G} \models EFp$ (resp. $\mathcal{G} \models EGp$) iff there exists $\rho \in Runs(\mathcal{G})$ s.t. $\rho = s^0 act^0 s^1 act^1 \ldots$ and $p \in \mathcal{L}(s^i)$ for some (resp. for all) $i \in \mathbb{N}$.*

By replacing $\models$ with $\models_F$ and *Runs* with $Runs_F$ in Definition 7 we obtain the notion of *one-shot* F-reachability and the dual of liveness. Both EF and EG are Computation Tree Logic (CTL) modalities [10]. If $\mathcal{G} \models EFp$, then we say that p is reachable in $\mathcal{G}$ from the initial state.

Definition 8. *Let $\mathcal{N} \subseteq Net$ and $\rho*$ be a prefix of some $\rho \in Runs(\mathcal{G})$ s.t. $\rho* = s^0 act^0 s^1 act^1 \ldots$ By $\rho*{\downarrow}(\mathcal{N})$ we denote the projection of $\rho*$ to the asynchronous product of LTSs in $\mathcal{N}$, i.e. the result of transforming $\rho*$ by (1) firstly, projecting each s^i on the LTSs in $\mathcal{N}$; (2) secondly, removing the actions that do not belong to $\bigcup_{\mathcal{M}\in\mathcal{N}} acts(\mathcal{M})$, together with their destinations.*

Intuitively, $\rho*{\downarrow}(\mathcal{N})$ contains the parts of the global states of $\rho*$ that belong to $||_{\mathcal{M}\in\mathcal{N}}\mathcal{M}$ and actions executed by some $\mathcal{M} \in \mathcal{N}$. It is not difficult to show that $\rho*{\downarrow}(\mathcal{N})$ does not need to be a valid run of $||_{\mathcal{M}\in\mathcal{N}}\mathcal{M}$.

Example 2. Consider a sequence: $\eta = (r_0, s_0, t_0)\ \tau\ (r_0, s_0, t_1)\ \tau\ (r_0, s_0, t_2)\ open\ (r_1, s_0, t_2)\ chooseR\ (r_4, s_0, t_0)\ \tau\ (r_4, s_0, t_1)\ chooseL\ (r_0, s_0, t_0)$. Here, we have $\eta{\downarrow}(\{\mathcal{R}, \mathcal{M}_1\}) = (r_0, s_0) open (r_1, s_0) chooseR (r_4, s_0) chooseL (r_0, s_0)$.

3 Compact Representations of State Spaces of Live-Reset and Sync-Deadlock Trees

In this section we show how to generate compact representations of state spaces of sync-memoryless tree topologies preserving reachability. The procedure is presented in two steps. We start with the case of two-level trees. Then, we modify the construction to deal with trees of arbitrary height in a bottom-up manner.

[3] The family of sync-memoryless LTSs can in the future be extended beyond these two classes.

3.1 Constructions for Two-level Trees

Throughout this subsection let $\mathcal{G}$ be a sync-memoryless tree $\mathcal{ST}$ with components $Net = \{\mathcal{R}, \mathcal{M}_1, \ldots, \mathcal{M}_n\}$ s.t. $root(\mathcal{G}) = \mathcal{R}$ and $children(\mathcal{R}) = \{\mathcal{M}_1, \ldots, \mathcal{M}_n\}$. Moreover, let $\mathcal{R} = \langle \mathcal{S}_\mathcal{R}, s^I_\mathcal{R}, Acts_\mathcal{R}, \rightarrow_\mathcal{R}, \mathcal{L}_\mathcal{R} \rangle$ and $\mathcal{M}_i = \langle \mathcal{S}_i, s^I_i, Acts_i, \rightarrow_i, \mathcal{L}_i \rangle$, for $i \in \{1, \ldots, n\}$. We employ the observations on the nature of synchronisations with sync-memoryless components in the following definition.

Definition 9 (Sum-of-squares Product). *Define* $\mathcal{SQ}^u(\mathcal{G}) = \langle \mathcal{S}^u_{sq}, s^I_{sq}, Acts_{sq}, \rightarrow_{sq}, \mathcal{L}_{sq} \rangle$ *as an* $\mathcal{LTS}$ *s.t.:*

1. $\mathcal{S}^u_{sq} = \bigcup_{i=1}^{n} (\mathcal{S}_i \times \mathcal{S}_\mathcal{R}) \cup \{s^I_{sq}\}$.
2. $s^I_{sq} \notin \mathcal{S}^u_{sq}$ *is a fresh initial state.*
3. $Acts_{sq} = acts(\mathcal{G}) \cup \{\epsilon\}$, *where* $\epsilon \notin acts(\mathcal{G})$ *is a fresh, silent action.*
4. *The transition relation* $\rightarrow_{sq}$ *is defined as follows:*
   (a) $s^I_{sq} \xrightarrow{\epsilon}_{sq} (s^I_i, s^I_\mathcal{R})$, *for all* $i \in \{1, \ldots, n\}$;
   (b) *if* $(s_i, s_R) \xrightarrow{act} (s'_i, s'_\mathcal{R})$ *is a transition in* $\mathcal{M}_i||\mathcal{R}$, *then also* $(s_i, s_\mathcal{R}) \xrightarrow{act}_{sq} (s'_i, s'_\mathcal{R})$, *for all* $i \in \{1, \ldots, n\}$;
   (c) *if* $\mathcal{G}$ *is live-reset and* $(s_i, s_\mathcal{R}) \xrightarrow{act} (s^I_i, s'_\mathcal{R})$ *is a transition in* $\mathcal{M}_i||\mathcal{R}$, *then* $(s_i, s_\mathcal{R}) \xrightarrow{act}_{sq} (s^I_j, s'_\mathcal{R})$ *for all* $j \in \{1, \ldots, n\} \setminus \{i\}$;
   (d) *if* $\mathcal{G}$ *is sync-deadlock and* $(s_i, s_\mathcal{R}) \xrightarrow{act} (s'_i, s'_\mathcal{R})$ *is a synchronised transition in* $\mathcal{M}_i||\mathcal{R}$, *then* $(s_i, s_\mathcal{R}) \xrightarrow{act}_{sq} (s^I_j, s'_\mathcal{R})$, *for all* $j \in \{1, \ldots, n\} \setminus \{i\}$.
5. $\mathcal{L}_{sq}(s_i, s_\mathcal{R}) = \mathcal{L}_i(s_i) \cup \mathcal{L}_\mathcal{R}(s_\mathcal{R}) \cup \bigcup_{j \neq i} \mathcal{L}_j(s^I_j)$, *for each* $(s_i, s_\mathcal{R}) \in \mathcal{S}^u_{sq}$.

We call $\mathcal{SQ}^u(\mathcal{G})$ *the Sum-of-squares Product of* $\mathcal{G}$.

Intuitively, $\mathcal{SQ}^u(\mathcal{G})$ at any given moment traces only the interactions between the root and one of its children. Item 4a of Definition 9 introduces the new initial state connected via ϵ-transitions with the initial states of each square product $\mathcal{M}_i||\mathcal{R}$. Item 4b ensures that the square product preserves the local and synchronised actions of the root and each child. Item 4c means that after resetting each component $\mathcal{M}_i$ can release control to another module, for live-reset topologies. Item 4d serves a similar purpose for sync-deadlock topologies.

Example 3. Fig. 2 presents the sum-of-squares product $\mathcal{SQ}^u(\mathcal{G}_x)$ for the small tree $\mathcal{ST}$ of Example 1. The fresh initial state s^I_{sq} is coloured green, the yellow box surrounds the square product $\mathcal{M}_1||\mathcal{R}$, and the pink box surrounds the square product $\mathcal{M}_2||\mathcal{R}$. The red colour of some states is explained in Section 3.2. The arcs that correspond to the transitions synchronised between a child and the root are coloured red if the control switches between components (according to Item 4c of Definition 9) and blue otherwise. The local transitions are black.

Theorem 1 (Sum-of-squares Product Preserves Reachability). *Let* $\mathcal{G}$ *be a sync-memoryless two-level tree* $\mathcal{ST}$ *with root* $\mathcal{R}$ *and* $p \in \mathcal{PV}$. *If* $\mathcal{G}$ *is live-reset, then* $\mathcal{G} \models EFp$ *iff* $\mathcal{SQ}^u(\mathcal{G}) \models EFp$. *If* $\mathcal{G}$ *is sync-deadlock and* $F = downacts(\mathcal{R})$, *then* $\mathcal{G} \models_F EFp$ *iff* $\mathcal{SQ}^u(\mathcal{G}) \models_F EFp$.

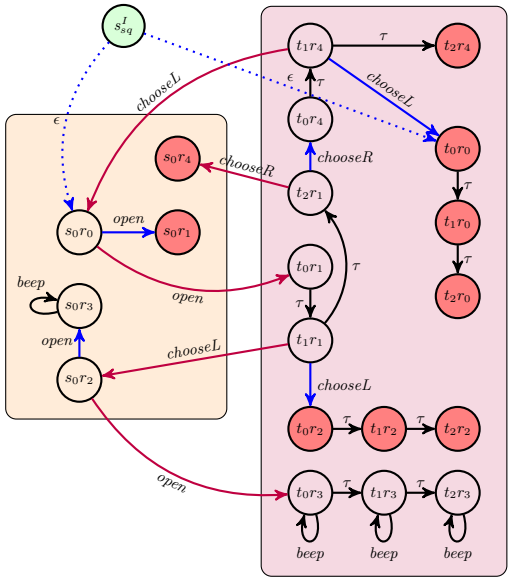

Fig. 2. The sum-of-squares product of the tree $\mathcal{ST}$ of Example 1.

Proof. We only deal with the case of live-reset topology; the case of sync-deadlock follows similarly.

Recall that we assume $Net = \{\mathcal{R}, \mathcal{M}_1, \ldots, \mathcal{M}_n\}$ with root $\mathcal{R}$ and children $\{\mathcal{M}_i\}_{i=1}^n$. Let $\mathcal{G} \models EFp$ and $\rho = s^0 act^0 s^1 act^1 \ldots$ be a run of $\mathcal{G}$ s.t. $p \in \mathcal{L}(s^i)$ for some $i \in \mathbb{N}$. If ρ contains only local actions, then for any of the components $C \in Net$ each of its local states visited along ρ can be reached by executing in $\mathcal{SQ}^u(\mathcal{G})$ firstly the ϵ-action (cf. Item 4a of Definition 9) and then local actions of C found in ρ (cf. Item 4b of of Definition 9). Moreover, consider a situation where a non-root component C synchronises with the root only a finite number of times. Now, each local state s of C that is reachable along ρ only after executing the final synchronising action of C can be reached in $\mathcal{SQ}^u(\mathcal{G})$ using only ϵ- and local actions. Such a run in $\mathcal{SQ}^u(\mathcal{G})$ reaching s can be built as described previously due to the fact that the final synchronisation resets the component. Also, let $\hat{\rho}$ be the result of iterative removal from ρ of all the local actions of any non-root component C that appear after its last synchronising action together with their targets, and replacing along the run all the further local states of C with s^I_C. It is not difficult to see that $\hat{\rho}$ visits the same local states of the root that are visited along ρ and all the local states of each non-root component C visited before the final synchronising action of C is executed.

We can therefore further assume ($\star$) that ρ contains at least one synchronising action and can be represented as $\rho = \alpha_1 F_1 \alpha_2 F_2 \ldots$, where for each $i \in \mathbb{N}$ there exist $j, k \in \mathbb{N}$ such that $\alpha_i = s^j act^j \ldots s^k act^k s^{k+1}$ and $act^j, \ldots, act^k \in locacts(\mathcal{R}) \cup \bigcup_{i=1}^n locacts(\mathcal{M}_i)$, $F_i \in downacts(\mathcal{R})$, and each local action of any component is eventually followed by some of its synchronising actions.

Actions are never synchronised between children, thus it can be proven by induction on the length of the run that the actions in ρ can be reordered to obtain a run $\rho' \in \mathit{Runs}(\mathcal{G}, s^0)$ that can be represented as $\rho' = \alpha'_1 F_1 \alpha'_2 F_2 \ldots$, such that:

1. For any $i \in \mathbb{N}$ there exist $j, k \in \mathbb{N}$ such that $\alpha'_i = s'^j \mathit{act}'^j \ldots s'^k \mathit{act}'^k s'^{k+1}$ and $\mathit{act}'^j, \ldots, \mathit{act}'^k \in \mathit{locacts}(\mathcal{R}) \cup \mathit{locacts}(\mathit{snd}(F_i))$.
2. For each $i \in \mathbb{N}$ we have $\alpha_i{\downarrow}\,(\{\mathcal{R}, \mathit{snd}(F_i)\}) = \alpha'_i{\downarrow}\,(\{\mathcal{R}, \mathit{snd}(F_i)\})$.
3. For each s'^j in α'_i, if 0 is the coordinate of root and k is the coordinate of $\mathit{snd}(F_i)$, then $s'^j = (s_0, s_1^I, \ldots, s_{k-1}^I, s_k, s_{k+1}^I, \ldots)$ for some $s_0 \in \mathit{states}(\mathcal{R})$, $s_k \in \mathit{states}(\mathit{snd}(F_i))$.

Intuitively, ρ' is built from ρ in such a way that firstly only the root and the component that synchronises with the root over F_1 are allowed to execute their local actions while all the other components stay in their initial states; then F_1 is fired; and then this scheme is repeated for F_2, F_3, etc. We can now project ρ' on spaces of squares of the root and components active in a given interval, to obtain $\rho'' = \alpha'_1{\downarrow}\,(\{\mathcal{R}, \mathit{snd}(F_1)\})F_1\alpha'_2{\downarrow}\,(\{\mathcal{R}, \mathit{snd}(F_2)\})F_2 \ldots$ Notice that $\rho'' \in \mathcal{SQ}^u(\mathcal{G})$. We now show that ρ'' visits all the local states that appear along ρ. To this end firstly observe that ρ and ρ' contain the same local states. Moreover, if a local state of the root is visited in ρ' then it is also visited in ρ'', as we always project on squares of the root and some other component. If a local state of a non-root component is visited along ρ' before executing some synchronising action F_i then it will also be present before executing F_i along ρ''. This follows from the construction of ρ'' from ρ' and our assumption ($\star$) of the structure of ρ. As we have shown that ρ'' visits each local state that appears along ρ, this part of the proof is concluded.

Let $\mathcal{SQ}^u(\mathcal{G}) \models EFp$ and $\rho \in \mathit{Runs}(\mathcal{SQ}^u(\mathcal{G}))$ visits a state labelled with p. Now, it suffices to replace in ρ each state (s_k, s_0) that belongs to the square $\mathcal{M}_k \times \mathcal{R}$ with the global state $(s_0, s_1^I, \ldots, s_{k-1}^I, s_k, s_{k+1}^I, \ldots)$ of $\mathcal{G}$. The result of this substitution is a run of $\mathcal{G}$ that visits p. □

The next proposition shows that the sum-of-squares does not preserve EG.

Proposition 1 (Sum-of-squares Does Not Preserve EG). *There is a live-reset two-level tree $\mathcal{ST}$ $\mathcal{G}$ s.t. for some $p \in \mathcal{PV}$, $\mathcal{G} \models EGp$ and $\mathcal{SQ}^u(\mathcal{G}) \not\models EGp$.*

Proof. Consider the tree $\mathcal{ST}$ $\mathcal{G}_y$ in Fig. 3. Here, we have $\mathcal{G}_y \models EGp$, but each path ρ along which p holds globally starts with $\mathcal{M}_1^y$ executing τ followed by $\mathcal{M}_2^y$ executing τ and, consecutively, *chooseR*. Thus, it is not possible to partition ρ into intervals where one child and the root fire local actions until they synchronise and possibly release the control to another child. Hence, $\mathcal{SQ}^u(\mathcal{G}_y) \not\models EGp$. □

As illustrated in Example 3, the size of the state space of the sum-of-squares product of a sync-memoryless $\mathcal{ST}$ $\mathcal{G}$ can be equal to or greater than the size of the state space of $\mathcal{G}$. On the other hand, the size of a representation of a state will be smaller in the sum-of-squares product, as it records only local states of at most two components of the network. However, in less degenerate cases than

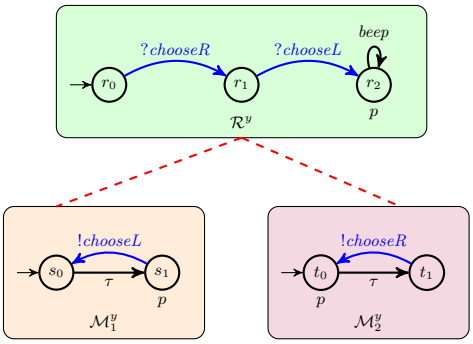

Fig. 3. The sum-of-squares does not preserve EG.

our toy model we can expect significant reductions. Let us consider a tree with a root and n children, each with a state space of size m. The number of states in the asynchronous product is m^{n+1}. In contrast, the size of the sum-of-squares product of such a topology is $n \cdot m^2 + 1$. Indeed we record pairs of states from a child and its root, i.e. m^2 possible states. This is done for all n children, and there is an added fresh initial state. To summarise, the sum-of-squares product has a size in $O(n \cdot m^2)$.

3.2 Reduced Sum-of-Squares for Any Tree Height

We now adapt the sum-of-squares product of two-level live-reset tree topologies to the general case. To this end we introduce the auxiliary operation *cmpl* that transforms $\mathcal{SQ}^u(\mathcal{G})$ into a live-reset LTS. Intuitively, *cmpl* redirects each transition that enters the initial state of the root to the fresh initial state s^I_{sq}. No additional operations are needed for sync-deadlock topologies.

Definition 10. *Let $\mathcal{G}$ be a live-reset two-level tree $\mathcal{ST}$ with components $Net = \{\mathcal{R}, \mathcal{M}_1, \ldots, \mathcal{M}_n\}$. By $cmpl(\mathcal{SQ}^u(\mathcal{G}))$ we denote the result of replacing in $\mathcal{SQ}^u(\mathcal{G})$ every transition $((s_{\mathcal{M}_i}, s_{\mathcal{R}}), act, (s'_{\mathcal{M}_i}, s^I_{\mathcal{R}}))$ with $((s_{\mathcal{M}_i}, s_{\mathcal{R}}), act, s^I_{sq})$.*

Algorithm 1 recursively performs reduction for two-level trees in a bottom-up manner. If the topology is live-reset, then each reduction is followed by applying $cmpl(\mathcal{SQ}^u(\cdot))$ to the computed sum-of-squares to ensure that the output is also live-reset. Note that $\mathcal{G}_{chld}$ denotes the subtree of $\mathcal{G}$ rooted in *chld*. The next theorem states soundness and correctness of Algorithm 1.

Theorem 2 ($reduceNet(\mathcal{G})$ Preserves Reachability). *Let $\mathcal{G} = \langle Net, \mathcal{T} \rangle$, be a sync-memoryless tree $\mathcal{ST}$, $F = \bigcup_{\mathcal{M} \in Net} downacts(\mathcal{M})$, and $p \in \mathcal{PV}$. If $\mathcal{G}$ is live-reset, then $\mathcal{G} \models EFp$ iff $reduceNet(\mathcal{G}) \models EFp$. If $\mathcal{G}$ is sync-deadlock, then $\mathcal{G} \models_F EFp$ iff $reduceNet(\mathcal{G}) \models_F EFp$.*

Proof. (Sketch) The proof follows via induction on the height of the tree $\mathcal{G}$. As we have Theorem 1, it suffices to prove that $cmpl(\mathcal{SQ}^u(\mathcal{G}))$ preserves reachability

for any two-level live-reset $\mathcal{ST}$ $\mathcal{G}$. This, however, can be done in a way very similar to the proof of Theorem 1 and is omitted.

Algorithm 1 *reduceNet*($\mathcal{G}$)

Input: **sync-memoryless tree sync. topology** $\mathcal{G}$
Output: $\mathcal{LTS}$ $\mathcal{M}$ that preserves reachability of each proposition p in $\mathcal{G}$.

```
1: if G consists of a single LTS then
2:    return G (* G is a leaf *)
3: end if
4: let redChdn := ∅
5: for chld ∈ children(root(G)) do
6:    redChdn = redChdn ∪ {reduceNet(G_chld)}
7: end for
8: let G' := {root(G)} ∪ redChdn
9: if G is live-reset then
10:   return cmpl(SQ^u(G'))
11: else
12:   return SQ^u(G')
13: end if
```

In certain hierarchical systems such as attack-defense trees [17] we are interested only in the reachability of the root's locations. This enables for additional optimisations related to removing deadlocks and livelocks that halt the root's evolution.

Definition 11 (Root-deadlock). *Let $\mathcal{G}$ be a $\mathcal{ST}$ with Net $= \{\mathcal{R}, \mathcal{M}_1, \ldots, \mathcal{M}_n\}$, root$(\mathcal{G}) = \mathcal{R}$ and children$(\mathcal{R}) = \{\mathcal{M}_1, \ldots, \mathcal{M}_n\}$. We say that a state s of $\mathcal{M}_i \in$ children$(\mathcal{R})$ is in a* root-deadlock *iff there is no run $\rho \in$ Runs$(\mathcal{M}_i, s)$ s.t. $\rho = s^0 act^0 s^1 act^1 \ldots$ with $act^i \in$ acts$(\mathcal{R})$, where $s^0 = s$, for some $i \in \mathbb{N}$.*

The set of root-deadlocked states of an $\mathcal{LTS}$ can be computed in polynomial time using either a model checker or conventional graph algorithms. These states can be removed without affecting the reachability of a location of the root.

Example 4. Let us consider the LTS $\mathcal{SQ}^u(\mathcal{G}_x)$ in Fig. 2. If it is a child of another LTS *parent*$(\mathcal{SQ}^u(\mathcal{G}_x))$ in a live-reset topology and *beep* $\in$ *upacts*$(\mathcal{SQ}^u(\mathcal{G}_x))$, then all the states coloured red are root-deadlocked. Fig. 4 displays the reduced sum-of-squares product of the topology.

Let us now evaluate the size of the state spaces for a tree of height h, considering each node has n children, each with a state space of size m. Such a tree has $\sum_{i=0}^{h} n^i$ nodes. Hence, the number of states in the asynchronous product is $m^{\sum_{i=0}^{h} n^i}$. The size of the sum-of-squares product of such a topology is $n^h \cdot m^{h+1} + \sum_{i=0}^{h-1} n^i \cdot m^i$. Indeed, for $h = 1$, this is exactly the size obtained for two-level trees in the previous section. The proof for any arbitrary

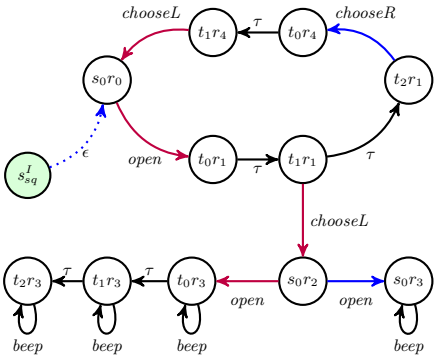

Fig. 4. The reduced sum-of-squares product of the tree $\mathcal{ST}$ of Example 4.

height is easily done by induction. To compute the increase from height h to height $h+1$, the same arguments as for the computation in a two-level tree hold. This leads to the following maximal number of states for height $h+1$: $1+n\cdot(n^h\cdot m^{h+1}+\sum_{i=0}^{h-1} n^i\cdot m^i)\cdot m = n^{h+1}.m^{h+2}+\sum_{i=0}^{h} n^i\cdot m^i$. To summarise, the sum-of-squares product has a size in $O(n^h\cdot m^{h+1})$.

4 The General Case and Local Products

The case of live-reset LTS tree networks entails substantial state space reductions due to the fact that children reset after synchronising with their root. Before resetting, only the local states of the root and one of its children need to be recorded; it can be assumed that all the children wait in their initial states. After single-level synchronisation, only the information about the root's state is relevant, as the synchronising child resets. In the general case of tree-like networks of synchronising LTSs, both the root's state and post-synchronisation state of its children have to be preserved.

In what follows let $\mathcal{G}$ be a two-level tree $\mathcal{ST}$ s.t. $Net = \{\mathcal{R}, \mathcal{M}_1, \ldots, \mathcal{M}_n\}$, where $root(\mathcal{G}) = \mathcal{R}$ and $children(\mathcal{R}) = \{\mathcal{M}_1, \ldots, \mathcal{M}_n\}$. Let $\mathcal{R} = \langle \mathcal{S}_\mathcal{R}, s^I_\mathcal{R}, Acts_\mathcal{R}, \rightarrow_\mathcal{R}, \mathcal{L}_\mathcal{R}\rangle$ and $\mathcal{M}_i = \langle \mathcal{S}_i, s^I_i, Acts_i, \rightarrow_i, \mathcal{L}_i\rangle$, for $i \in \{1, \ldots, n\}$. For each $i \in \{1, \ldots, n\}$ let $postSync(\mathcal{M}_i)$ denote the set of all the local states of $\mathcal{M}_i$ visited immediately after synchronising with $\mathcal{R}$. Formally:

$$postSync(\mathcal{M}_i) = \{s \in \mathcal{S}_i \mid \exists act \in Acts_\mathcal{R} \cap Acts_i \text{ s.t. } (s', r') \xrightarrow{act} (s, r) \text{ in } \mathcal{S}_i \times \mathcal{S}_\mathcal{R}\}.$$

The *memory unit*, defined as follows, is a collection of all the vectors of states of the components preserved after synchronisation.

Definition 12 (Memory Unit). *The* post-synchronisation memory unit *of* $\mathcal{G}$ *is defined as:* $mem(\mathcal{M}_1, \ldots, \mathcal{M}_n) = \prod_{i=1}^{n}(\{s^I_i\} \cup postSync(\mathcal{M}_i))$.

Intuitively, Definition 13 generalises Definition 9 by extending the states of square products with the memory unit that is updated and consulted whenever a

synchronisation takes place. This memory gadget can be implemented efficiently, but here we present it in an explicit way for the sake of readability.

Let $i \in \{1,\dots,n\}$, $s_i \in \mathcal{S}_i$, and $m \in mem(\mathcal{M}_1,\dots,\mathcal{M}_n)$. By $m[i/s_i]$ we denote the *memory update* of m defined as follows: $m[j] = m[j]$ for all $i \neq j$ and $m[i] = s_i$.

Definition 13 (General Sum-of-squares Product). *Let* $\mathcal{GSQ}(\mathcal{G}) = \langle \mathcal{S}_{gsq}, s^I_{gsq}, Acts_{gsq}, \to_{gsq}, \mathcal{L}_{gsq}\rangle$ *be an* $\mathcal{LTS}$ *s.t.:*

1. $\mathcal{S}_{gsq} = \big(\bigcup_{i=1}^{n}(\mathcal{S}_i \times \mathcal{S}_{\mathcal{R}}) \times mem(\mathcal{M}_1,\dots,\mathcal{M}_n)\big) \cup \{s^I_{gsq}\}$.
2. $s^I_{gsq} \notin \mathcal{S}_{gsq}$ *is a fresh initial state.*
3. $Acts_{gsq} = acts(\mathcal{G}) \cup \{\epsilon\}$, *where* $\epsilon \notin acts(\mathcal{G})$ *is a fresh, silent action.*
4. *The transition relation* $\to_{gsq}$ *is defined as follows:*
   (a) *For each* $i \in \{1,\dots,n\}$, $s^I_{gsq} \xrightarrow{\epsilon}_{gsq} (s^I_i, s^I_{\mathcal{R}}, m_0)$, *where* $\forall^n_{i=1} m_0[i] = s^I_i$.
   (b) *If* $s_i \xrightarrow{act}_i s'_i$ *and* $act \in locacts(\mathcal{M}_i)$, *then* $(s_i, s_{\mathcal{R}}, m) \xrightarrow{act}_{sq} (s'_i, s_{\mathcal{R}}, m)$, *for each* $s_{\mathcal{R}} \in \mathcal{S}_{\mathcal{R}}$ *and* $m \in mem(\mathcal{M}_1,\dots,\mathcal{M}_n)$; *similarly, if* $s_{\mathcal{R}} \xrightarrow{act}_{\mathcal{R}} s'_{\mathcal{R}}$ *and* $act \in locacts(\mathcal{R})$, *then* $(s_i, s_{\mathcal{R}}, m) \xrightarrow{act}_{sq} (s_i, s'_{\mathcal{R}}, m)$, *for each* $s_i \in \mathcal{S}_i$ *and* $m \in mem(\mathcal{M}_1,\dots,\mathcal{M}_n)$.
   (c) *If* $act \in upacts(\mathcal{M}_i)$, $s_i \xrightarrow{act}_i s'_i$, *and* $s_{\mathcal{R}} \xrightarrow{act}_{\mathcal{R}} s'_{\mathcal{R}}$, *then* $(s_i, s_{\mathcal{R}}, m) \xrightarrow{act}_{sq} (s_j, s'_{\mathcal{R}}, m')$, *where* $m' = m[i/s'_i]$ *and* $s_j = m[j]$ *for some* $j \in \{1,\dots,n\}$.
5. $\mathcal{L}_{gsq}(s_i, s_{\mathcal{R}}, m) = \mathcal{L}_i(s_i) \cup \mathcal{L}_{\mathcal{R}}(s_{\mathcal{R}}) \cup \bigcup_{j\neq i} \mathcal{L}_j(m[j])$, *for each* $(s_i, s_{\mathcal{R}}, m) \in \mathcal{S}_{gsq}$.

We call $\mathcal{GSQ}(\mathcal{G})$ *the General Sum-of-squares Product of* $\mathcal{G}$.

Item 4a of Definition 13 expresses that the new initial state enables ϵ-transitions to the initial state of any square product $\mathcal{M}_i||\mathcal{R}$ with the memory unit set to the starting values. Similarly to the corresponding case in Definition 9, Item 4b ensures that the local actions are fully asynchronous and do not affect memory unit. The idea behind Item 4c is that in a state $(s_i, s_{\mathcal{R}}, m)$ executing a synchronised action *act* will require updating the outcome states, saving them into memory, and then moving all the values from memory to the current state tracker. The latter may mean switching to another component. The transition rule, as defined here, concatenates these three steps into one.

Example 5. Fig. 5 presents a three-component $\mathcal{ST}$ $\mathcal{G}_z$. Note that the child $\mathcal{M}^z_2$ is not live-reset, as it does not reset after synchronising with $\mathcal{R}^z$ via *chooseF*. Fig. 6 presents the general sum-of-squares product of $\mathcal{G}_z$.

The following analogues of Theorem 1 and Theorem 2 show that the general sum-of-squares preserves reachability. We omit the proofs of Theorem 3 and Theorem 4, as they follow via almost exactly the same techniques.

Theorem 3 (General sum-of-squares Preserves Reachability). *Let* $\mathcal{G}$ *be a two-level tree* $\mathcal{ST}$. *For each* $p \in \mathcal{PV}$ *we have* $\mathcal{G} \models EFp$ *iff* $\mathcal{GSQ}(\mathcal{G}) \models EFp$.

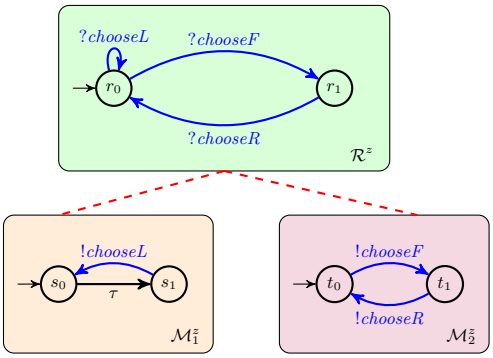

Fig. 5. A simple non live-reset tree synchronisation topology $\mathcal{G}_z$

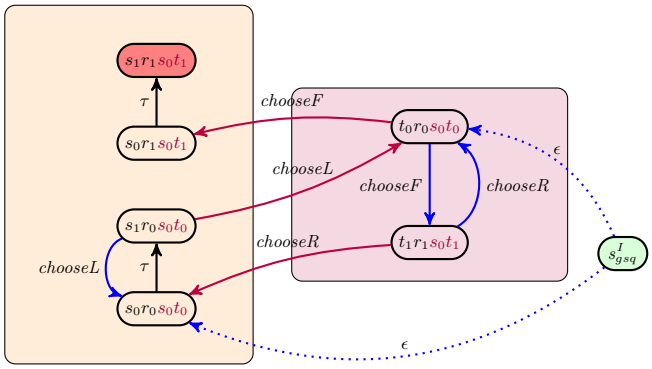

Fig. 6. The general sum-of-squares product of the non live-reset tree $\mathcal{ST}$ in Fig. 5

A two-level general sum-of-squares can be adapted to deal with tree topologies $\mathcal{G}$ of any height by using recursive construction. The appropriate algorithm, denoted by *reduceGenNet*($\mathcal{G}$), is a slight modification of Algorithm 1 with recursive call in Line 6 replaced with *reduceGenNet*($\mathcal{G}_{chld}$) and the if-else conditional in Line 9 substituted with **return** $\mathcal{GSQ}(\mathcal{G}')$.

Theorem 4 (*reduceGenNet*($\mathcal{G}$) **Preserves Reachability).** *Let $\mathcal{G}$ be a tree $\mathcal{ST}$. For each $p \in \mathcal{PV}$, we have $\mathcal{G} \models EFp$ iff reduceGenNet$(\mathcal{G}) \models EFp$.*

It can be also easily observed that $\mathcal{GSQ}(.)$ does not preserve EG, by the same argument as in Proposition 1 and Fig. 3. Moreover, as previously, root-deadlocked states (e.g. the red location in Fig. 6) can be removed if only the reachability of a location in the root is to be preserved.

The general sum-of-squares does not guarantee a reduction of the statespace size. If $\mathcal{G}$ is a two-level tree $\mathcal{ST}$ with $Net = \{\mathcal{R}, \mathcal{M}_1, \ldots, \mathcal{M}_n\}$, then the size of the statespace of *reduceGenNet*($\mathcal{G}$) can reach $\sum_{i=1}^{n} |\mathcal{S}_i| \cdot |\mathcal{S}_\mathcal{R}| \cdot \prod_{i=1}^{n} |\mathcal{S}_i|$, thus it

crucially depends on the size of the memory unit. On the other hand, we conjecture that the memory needed to preserve the general sum-of-squares product is often much smaller than the memory needed to hold the asynchronous product of the entire network. This conjecture is based on the observation that the states of $\mathcal{GSQ}(\mathcal{G})$ are composed of two parts: the pair of local states of two interacting modules and the memory unit which can be shared when implemented efficiently.

5 Experiments

In this section we evaluate the implementation of reductions for sync-memoryless tree topologies. The files to reproduce our tests and figures can be found at https://depot.lipn.univ-paris13.fr/parties/publications/live-trees.

The theory presented in Section 3 has been implemented in the open-source tool `LTR` [2], written in C. `LTR` accepts LTSs networks in a *modgraph* format [16]. The size of the fully synchronised product is computed using a Binary Decision Diagrams-based open-source Python tool `DD-Net-Checker` [1].

Model Generators: Attack-Defence Trees Attack-Defence Trees (ADTs) [15] are graphical models for representing possible scenarios of incoming risks and methods for their mitigation for complex systems. While descending from informal models, ADTs have been extended with various semantics. Here, we re-use the semantics based on translating the ADTs to networks of communicating LTSs [17,7]. These networks form tree-like synchronisation topologies with all the LTSs being sync-deadlock. An attack is deemed a success if a special location in the root of a network is reachable. For the purpose of this paper we implemented a simple translator from ADTs to modgraph format.

Comparing with *ADT* Reductions In [17] we presented techniques for simplifying tree-like networks using *pattern-* and *layered* reductions. The former are similar to partial order reductions and the crux of latter is in the observation that it is sufficient to consider only runs where each level of a tree fully synchronises with its children before the execution proceeds to a higher level. In [17] these techniques are implemented using time parameter injection into LTSs networks and translation to timed LTSs networks.

The comparison with the results of reductions from [17] is included for reference, as the cited work is aimed at the full timed LTS-based semantics of ADTs which involves numeric attributes such as the time and cost of attack. Under this semantics the networks produce considerably larger fully synchronised models. Therefore, the comparison may be slightly unfair, favouring our approach.

***ADT* Experiments** Table 1 shows the evaluation of the experiments on scalable models from [17]. Table 2 presents the results of running the experiments on security case studies, also taken from [17]. It should be noted that for the

Table 1. Results for scalable models from [17].

model	no reductions		sos reduction		pattern		pattern+layer		pattern+sos		reduced/original model size (%)			
	$\|S\|$	$\|T\|$	$\|S\|$	$\|T\|$	$\|S\|$	$\|T\|$	$\|S\|$	$\|T\|$	$\|S\|$	$\|T\|$	sos reduction	pattern	pattern+layer	pattern+sos
(2, 7, 2, 4)	185	432	111	310	72	123	54	69	45	62	68.233 %	31.605 %	19.935 %	17.342 %
(2, 9, 3, 4)	587	1,698	561	1,828	246	571	190	373	107	192	104.551 %	35.755 %	24.639 %	13.085 %
(2, 13, 3, 6)	8,823	35,602	861	2,800	2,405	7,584	883	1,802	176	306	8.241 %	22.485 %	6.044 %	1.085 %
(2, 15, 3, 8)	34,481	160,096	1,111	3,630	6,734	23,135	1,808	3,439	239	417	2.437 %	15.351 %	2.697 %	0.337 %
(2, 11, 4, 4)	1,825	6,332	2,811	10,498	840	2,458	652	1,680	231	513	163.16 %	40.432 %	28.589 %	9.121 %
(2, 15, 4, 6)	26,725	124,708	4,311	16,078	8,184	30,773	3,256	9,167	393	866	13.464 %	25.726 %	8.204 %	0.831 %
(2, 17, 4, 8)	103,955	549,762	5,561	20,828	22,854	92,393	6,606	17,615	541	1,203	4.037 %	17.629 %	3.705 %	0.267 %
(2, 23, 4, 10)	5,417,613	37,414,404	7,111	26,550	783,271	4,282,992	71,965	237,154	709	1,536	0.079 %	11.828 %	0.722 %	0.005 %
(2, 13, 5, 4)	5,603	22,774	14,061	59,248	2,868	10,124	2,228	7,088	480	1,281	258.34 %	45.784 %	32.829 %	6.206 %
(2, 17, 5, 6)	80,687	428,086	21,561	90,748	27,926	121,887	11,258	38,693	827	2,203	22.074 %	29.446 %	9.818 %	0.596 %
(2, 19, 5, 8)	312,889	1,858,220	27,811	117,498	77,948	362,276	22,808	74,986	1,144	3,075	6.693 %	20.276 %	4.504 %	0.194 %
(2, 25, 5, 10)	16,261,031	123,086,630	35,561	149,828	2,645,472	16,058,689	273,484	1,128,571	1,537	4,060	0.133 %	13.423 %	1.006 %	0.004 %
(2, 15, 6, 4)	17,065	79,784	TO	TO	9,792	40,476	7,608	28,796	979	3,067		51.903 %	37.588 %	4.178 %
(2, 19, 6, 6)	243,085	1,446,656	TO	TO	95,336	473,670	38,520	155,698	1,696	5,313		33.674 %	11.494 %	0.415 %
(2, 21, 6, 8)	940,715	6,202,486	TO	TO	266,084	1,397,360	78,020	303,724	2,351	7,425		23.287 %	5.344 %	0.137 %
(2, 27, 6, 10)	48,799,477	401,798,336	TO	TO	9,015,346	60,176,445	TO	TO	3,193	9,937		15.356 %		0.003 %
(2, 17, 7, 4)	51,707	273,994	TO	TO	33,432	158,372	25,976	113,996	1,978	7,139		58.89 %	42.976 %	2.799 %
(2, 21, 7, 6)	731,303	4,828,186	TO	TO	325,492	1,813,652	131,564	611,188	3,435	12,403		38.477 %	13.36 %	0.285 %
(2, 23, 7, 8)	2,826,241	20,492,984	TO	TO	908,440	5,319,108	266,464	1,198,156	4,766	17,333		26.706 %	6.281 %	0.095 %
(2, 19, 8, 4)	156,145	926,420	TO	TO	114,144	609,608	88,688	442,736	3,977	16,283		66.855 %	49.089 %	1.871 %
(2, 23, 8, 6)	2,198,005	15,951,260	TO	TO	1,111,296	6,862,916	449,216	2,357,964	6,914	28,323		43.937 %	15.467 %	0.194 %
(2, 21, 9, 4)	470,483	3,093,598	TO	TO	389,712	2,316,544	302,800	1,694,352	7,976	36,571		75.931 %	56.036 %	1.25 %
(2, 23, 10, 4)	1,415,545	10,225,856	TO	TO	1,330,560	8,712,240	TO	TO	15,975	81,147		86.268 %		0.834 %
(3, 10, 2, 6)	3,803	15,598	316	1,515	289	654	250	537	71	124	9.438 %	4.861 %	4.056 %	1.005 %
(3, 13, 2, 9)	43,387	228,362	463	2,226	1,283	3,424	653	1,192	104	183	0.99 %	1.732 %	0.679 %	0.106 %
(3, 13, 3, 6)	34,739	186,530	2,227	12,018	1,563	4,700	1,380	4,025	168	400	6.438 %	2.83 %	2.443 %	0.257 %
(3, 16, 3, 9)	392,531	2,577,950	3,256	17,625	6,771	23,024	3,846	10,667	246	602	0.703 %	1.003 %	0.489 %	0.029 %
(3, 16, 4, 6)	314,699	2,097,686	15,604	93,729	8,511	31,912	7,530	27,559	364	1,102	4.532 %	1.676 %	1.455 %	0.061 %
(3, 19, 4, 9)	3,540,971	27,920,114	22,807	137,388	36,777	152,390	21,117	74,504	531	1,655	0.509 %	0.601 %	0.304 %	0.007 %
(3, 19, 5, 6)	2,840,483	22,663,754	TO	TO	46,377	208,290	41,040	180,639	758	2,803		0.999 %	0.869 %	0.014 %
(3, 22, 5, 9)	31,901,507	293,805,446	TO	TO	200,349	978,834	115,164	491,799	1,103	4,193		0.362 %	0.186 %	0.002 %
(3, 22, 6, 6)	25,597,115	238,092,350	TO	TO	252,729	1,322,490	223,650	1,150,257	1,548	6,799		0.597 %	0.521 %	0.003 %
(3, 25, 6, 9)	287,244,635	3,027,198,170	TO	TO	1,091,763	6,143,676	TO	TO	2,249	10,130		0.218 %		0.0 %
(3, 25, 7, 6)	230,505,107	2,450,127,602	TO	TO	1,377,243	8,228,232	TO	TO	3,130	15,979		0.358 %		0.001 %

Table 2. Results for security case studies from [17].

model	no reductions		sos reduction		pattern		pattern+layer		pattern+sos		reduced/original model size (%)			
	$\|S\|$	$\|T\|$	$\|S\|$	$\|T\|$	$\|S\|$	$\|T\|$	$\|S\|$	$\|T\|$	$\|S\|$	$\|T\|$	sos reduction	pattern	pattern+layer	pattern+sos
forestall	62,689	185,944	2,427	7,564	5,784	17,285	1,845	2,721	664	1,656	4.018 %	9.278 %	1.836 %	0.933 %
gain_admin	51,158,719	364,218,554	8,626	33,595	1,327,546	7,776,327	52,923	94,570	844	1,965	0.01 %	2.192 %	0.036 %	0.001 %
iot_dev	3,381	6,860	4,173	11,852	907	2,154	371	450	1,315	3,144	156.479 %	29.89 %	8.017 %	43.541 %
treasure_hunters	479	1,326	316	937	157	340	74	89	134	228	69.418 %	27.535 %	9.03 %	20.055 %

latter we used slightly simpler models than in [17], as contrary to the former paper our tool does not handle data variables such as cost, etc. The goal in each of these scenarios is reachability of a certain location in the root node and all the networks are sync-deadlock. The timeout was set to 30 minutes (displayed as `TO` in the tables).

The model signature in the first column of Table 1 consists of the branching factor per an ADT node, the total number of nodes, the depth and the width of an ADT [17]. In both tables the second pair of columns collects the details of unreduced models, the third collects the results of applying the sum-of-squares (here, abbreviated to *sos*) construction to the unreduced models, the fourth of applying only pattern reduction, the fifth of pipelining the pattern and layer reductions, and the sixth of pipelining the pattern reduction and the sum-of-squares. The remaining columns collect the relative reduction/blowup rates.

The experimental data suggests that the reductions for sync-memoryless networks, proposed in this paper, are often comparable or exceeding [17]. This is especially evident when the sum-of-squares is applied to pattern-reduced models. The nature of the sum-of-squares construction can lead to statespace blowup, but this seems observable only for smaller networks. Moreover, larger networks enable more relative reductions.

To assess this further we conducted an independent series of scalable experiments on random live-reset tree networks. We generated 210 live-reset tree networks of depths 1–3 and computed their (reduced) sum-of-squares products. Fig. 7 presents the scatterplot of the results. The red line denotes no reduction.

The same phenomenon as for sync-deadlock trees can be observed: the degree of reduction increases with the size of the network.

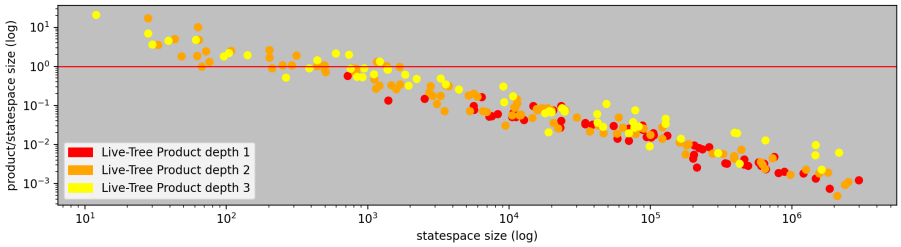

Fig. 7. Statespace sizes of sum-of-squares for live-reset tree networks.

6 Conclusion

In this paper we outlined how to simplify large tree networks of LTSs in which the components reset or deadlock after synchronising with their parents. We

also proposed and investigated a similar construction for the general tree-like synchronisation topologies. It is shown that the constructions preserve a certain form of reachability, but do not preserve liveness. An experimental evaluation shows that the method yields extremely effective reductions for sync-memoryless networks.

We raise several questions to be explored as future work. Firstly, we only have a very rough theoretical estimate of the size of reductions for sync-memoryless networks. Stronger estimations can be obtained. Secondly, for general networks we only put a hypothesis that the state space of the general sum-of-squares may substantially grow as compared to the full asynchronous product (albeit with possibly smaller memory usage). This can be investigated experimentally. Moreover, the "vanilla" technique of general sum-of-squares is straightforward, thus surely enables many optimisations. Thirdly, the class of live-reset tree networks is probably one of many that do not need tracing what happens after synchronisation (sync-deadlock networks are slightly different, as one-shot transitions still need to be traced). Other such classes and topologies could probably be identified. Finally, the sum-of-squares for sync-deadlock LTS networks guarantees that a state is reachable before reduction iff it is *one-shot* F-reachable after (see Definition 7 and Theorem 1). We however do not know the complexity of verifying *one-shot* F-reachability; here we conjecture that it is NP-hard, which may make the reductions for this class of models less impressive than suggested by experimental results. All of these concerns can be addressed in further research.

References

1. DD-Net-Checker. https://github.com/MichalKnapik/dd-net-checker (2021)
2. LTR. https://github.com/MichalKnapik/automata-net-reduction-tool (2021)
3. van der Aalst, W.M.P., van Hee, K.M.: Workflow Management: Models, Methods, and Systems. Cooperative information systems, MIT Press (2002)
4. Aminof, B., Kupferman, O., Murano, A.: Improved model checking of hierarchical systems. Inf. Comput. **210**, 68–86 (2012)
5. André, É.: IMITATOR 3: Synthesis of timing parameters beyond decidability. In: CAV. LNCS, vol. 12759, pp. 552–565. Springer (2021)
6. André, É., Lime, D., Ramparison, M., Stoelinga, M.: Parametric analyses of attack-fault trees. Fundam. Informaticae **182**(1), 69–94 (2021)
7. Arias, J., Budde, C.E., Penczek, W., Petrucci, L., Sidoruk, T., Stoelinga, M.: Hackers vs. security: Attack-defence trees as asynchronous multi-agent systems. In: ICFEM. LNCS, vol. 12531, pp. 3–19. Springer (2020)
8. Arias, J., Celerier, J.M., Desiante-Catherine, M.: Authoring and automatic verification of interactive multimedia scores. Journal of New Music Research (2016)
9. Arias, J., Petrucci, L., Masko, L., Penczek, W., Sidoruk, T.: Minimal schedule with minimal number of agents in attack-defence trees. In: ICECCS. pp. 1–10. IEEE (2022)
10. Baier, C., Katoen, J.: Principles of model checking. MIT Press (2008)
11. Behrmann, G., David, A., Larsen, K.G., Håkansson, J., Pettersson, P., Yi, W., Hendriks, M.: UPPAAL 4.0. In: QEST. pp. 125–126. IEEE Computer Society (2006)

12. Belardinelli, F., Lomuscio, A., Murano, A., Rubin, S.: Verification of broadcasting multi-agent systems against an epistemic strategy logic. In: IJCAI. pp. 91–97 (2017)
13. Holzmann, G.J.: The SPIN Model Checker - primer and reference manual. Addison-Wesley (2004)
14. Knapik, M., Meski, A., Penczek, W.: Action synthesis for branching time logic: Theory and applications. ACM Trans. Embed. Comput. Syst. **14**(4), 64:1–64:23 (2015)
15. Kordy, B., Mauw, S., Radomirovic, S., Schweitzer, P.: Attack-defense trees. J. Log. Comput. **24**(1), 55–87 (2014)
16. Lakos, C., Petrucci, L.: Modular analysis of systems composed of semiautonomous subsystems. In: ACSD. pp. 185–196. IEEE Computer Society (2004)
17. Petrucci, L., Knapik, M., Penczek, W., Sidoruk, T.: Squeezing state spaces of (attack-defence) trees. In: ICECCS. pp. 71–80. IEEE (2019)