2018 23rd International Conference on Engineering of Complex Computer Systems (ICECCS)

Extended Algebraic State-Transition Diagrams

Lionel N. Tidjon[1,2], Marc Frappier[1], Michael Leuschel[3], Amel Mammar[2]
[1] Université de Sherbrooke, 2500 boul. de l'Université, Sherbrooke (Québec), J1K 2R1, Canada
{lionel.nganyewou.tidjon,marc.frappier}@usherbrooke.ca,
[2]SAMOVAR, Télécom SudParis, CNRS, 9 rue Charles Fourier 91011 ÉVRY, France
amel.mammar@telecom-sudparis.eu,
[3]Institut für Informatik, Universität Düsseldorf, Düsseldorf, Germany
michael.leuschel@hhu.de

***Abstract*—Algebraic State-Transition Diagrams (ASTDs) are extensions of common automata and statecharts that can be combined with process algebra operators like sequence, choice, guard and quantified synchronization. They were previously introduced for the graphical representation, specification and proof of information systems. In an attempt to use ASTDs to specify cyber attack detection, we have identified a number of missing features in ASTDs. This paper extends the ASTD notation with state variables (attributes), actions on transitions, and a new operator called flow which corresponds to AND states in statecharts and is a compromise between interleaving and synchronization in process algebras. We provide a formal structured operational semantics of these extensions and illustrate its implementation in an OCaml-based interpreter called iASTD and the model checker ProB. Extended ASTDs are illustrated in a case study in cyber attack detection.**

***Index Terms*—Formal Methods, Algebraic State-Transition Diagrams, Information Systems, Cyber Attack Detection.**

I. Introduction

Model-based languages like Abstract State Machines (ASM) [1], B [2] and Z [3] provide rich environments for specifying data and behavioural aspects of systems. However, the modelling of control flow in these languages, e.g., sequence, choice or parallel composition, is not easy to understand and validate with users. Process algebras like Communicating Sequential Processes (CSP) [4] and the Calculus of Communicating Systems (CCS) [5] describe interactions, communications, and synchronizations between processes. They support rich operators like sequence, choice, parallel composition, interleaving and iteration. But, they do not support a concise and elegant way to describe complex data aspects [6].

Combinations of CSP and Z (e.g., Circus [7]), CSP and B (e.g. CSP||B [8], CSP2B [9]) integrate process algebras with model-based notations to provide a richer specification environment that provides a more explicit representation of the control flow and support a rich notation for data modelling. On the other hand, graphical notations like statecharts [10], [11] and their variants offer an even more explicit representation of control and have shown their usefulness in various domains. Algebraic-State Transition Diagrams (ASTDs) were proposed in [12] to combine the graphical strengths of statecharts and the abstraction power of process algebra operators. An elementary ASTD is an automaton. Elementary ASTDs can be easily integrated with operators like sequence, Kleene closure, choice, guard, parallel composition with synchronization, quantified choice and quantified parallel composition. Like in statecharts, automaton states can themselves be complex ASTDs. ASTDs can be translated into B for formal analysis and proof [13]. They can be associated with B machines to model data and they can be refined in the Event-B style by adding new events [14], [15]. A B machine contains one operation for each event and it encapsulates data associated to the ASTD, similar to the CSP2B approach. When an event is executed by the ASTD, the corresponding operation in the data B machine is also executed.

In this paper, we propose an extension of the ASTD notation to support the declaration of attributes (i.e., state variables) and actions that can modify these attributes when a transition is executed. In contrast to [8], [9], [15], these extensions enable to support data handling directly within the ASTD specification, closer to the statecharts style. Thus, the proposed extension provides the same level of data modelling as in tools like Stateflow [16], but with the additional control flow abstraction capability of process algebra operators. Attributes can be locally declared within each ASTD. Actions can be executed on automaton transitions, but also at the level of an ASTD itself, in order to easily factor out code that needs to be executed for every transitions of an ASTD. This paper proposes also a new ASTD operator called *flow* which corresponds to an AND state in statecharts and is a compromise between interleaving and synchronization in process algebras. It combines multiple ASTDs and executes an event on each of them that can execute it. Thus, it is a form of *weak* synchronization. We have identified the need for these extensions while exploring the use of ASTDs to model cyber security attacks.

This paper provides a formal operational semantics of these extensions. It also presents an interpreter implemented in OCaml and a model checker based on ProB [17] that can be used to execute and validate ASTD specifications.

The rest of this paper is structured as follows. Section II

978-1-5386-9341-4/18/$31.00 ©2018 IEEE
DOI 10.1109/ICECCS2018.2018.00023

introduces the extended ASTD notation and describes its operational semantics. Section III presents a case study in cyber attack detection and illustrates the proposed extensions. In Section IV, we present the ASTD interpreter and the model checker based on ProB. Section V concludes by some discussions and perspectives.

II. Extended ASTD Syntax and Semantics

This section aims to formally specify the extensions (i.e., attributes, actions and the new operator flow). Actions and attributes are required to more precisely and concisely describe cyber attacks [18], while the new operator allows executing multiple attack events from various sources (e.g., network, host) and combining different attack models, being parts of a whole attack. This is particularly useful to detect multi-step attacks and advanced persistent threats that originate from multiple entry points and attack vectors [19]. Introducing attributes and actions require adding a concept of global state to the ASTD semantic rules, while the new operator "simply" requires its own rules to be added, albeit with a twist (Sect. II-D).

The structure (syntax) of ASTDs is defined using a type hierarchy. Each ASTD operator is represented by a type. To identify characteristics shared by all ASTD types, we define an abstract ASTD type $\mathsf{ASTD} \triangleq \langle n, P, V, A_{astd} \rangle$ where $n \in \mathsf{Name}$ is the name of the ASTD, P is an optional list of parameters, V is a set of attributes, $A_{astd} \in \mathcal{A}$ is an action; its default value is **skip**, which does nothing. Parameters P are used to receive values passed by a calling ASTD; they can be read-only or read-write. Attributes V are state variables that can be modified by actions and tested in guards within the scope of the ASTD. Actions can also modify attributes received as parameters of the ASTD. Each ASTD type inherits from the supertype ASTD.

We also distinguish between the syntax of an ASTD and its state. We denote by States the set of states of ASTDs. Each ASTD type gives rise to a subtype of States. In this paper, we content ourselves with the definition of the following subtypes of ASTDs: elem, Sequence, Automaton, Synchronization and Flow. A complete definition of extended ASTDs is available at [20]. Final states of an ASTD are determined by a function *final* of type $\mathsf{ASTD} \times \mathsf{States} \rightarrow \mathsf{Boolean}$. Function *init* of type $\mathsf{ASTD} \rightarrow \mathsf{States}$ returns the initial state of an ASTD.

The semantics of ASTDs consists of a labeled transition system (LTS). A LTS is a subset of $\mathsf{States} \times \mathsf{Event} \times \mathsf{States}$ and a transition is denoted by $s \xrightarrow{\sigma}_a s'$. It means that ASTD a can execute event σ from state s and move to state s'. The semantics of an ASTD depend on the variables declared in its enclosing ASTDs; we use environments to represent the values of these variables. An environment is a partial function of type $\mathsf{Env} \triangleq \mathsf{Var} \nrightarrow \mathsf{Term}$ which assigns values to variables. We need to introduce an auxiliary transition relation that handles environments:

$$s \xrightarrow{\sigma, E_e, E'_e}_a s'$$

where E_e, E'_e denote the before and after values of variables in the ASTDs enclosing ASTD a. The first rule provided below relates the state-transition relation with the auxiliary one. It states that a transition is proved starting with empty environments.

$$\mathsf{env}\ \frac{s \xrightarrow{\sigma,\{\},\{\}}_a s'}{s \xrightarrow{\sigma}_a s'}$$

ASTDs are *non-deterministic*. If several transitions on σ are possible from a given state s, then one of them is non-deterministically chosen. The operational semantics is inductively defined in the sequel for some ASTD subtypes.

A. Automaton

1) Syntax: An automaton ASTD is a structure $\langle \mathsf{aut}, \Sigma, S, \nu, \delta, SF, DF, n_0 \rangle$ with the following constraints. $\Sigma \subseteq \mathsf{Event}$ is the alphabet. $S \subseteq \mathsf{Name}$ is the set of state names. $\nu \in S \rightarrow \mathsf{ASTD}$ maps each state name to its sub-ASTD, which can be elementary (noted elem) or complex. An automaton transition between states $n_1, n_2 \in S$ labeled with $\sigma[g]/A_{tr}$ is represented as a tuple in the transition relation δ as follows:

$$((\mathsf{loc}, n_1, n_2), \sigma, g, A_{tr}, \mathit{final}?) \in \delta$$

Symbol *final*? is a Boolean: when *final*? = true, the source of the transition is decorated with a bullet; it indicates that the transition can be fired only if n_1 is final. The *final*? field is only useful when n_1 is a complex state. We also write $\delta((\mathsf{loc}, n_1, n_2), \sigma, g, A_{tr}, \mathit{final}?)$ to state that a tuple is an element of δ.

There are other types of automaton transitions (e.g., to, or from, a state of a nested automaton); they are omitted here; see [20] for further information. $SF \subseteq S$ is the set of shallow final states, while $DF \subseteq S$ denotes the set of deep final states, with $DF \cap SF = \varnothing$. $n_0 \in S$ is the name of the initial state. The definition of $final$, provided in the sequel, will describe the distinction between the two types of final states.

The state of an automaton cannot be simply represented by a state name. It is a more complex structure of type $\langle \mathsf{aut}_\circ, n, E, s \rangle$. $\mathsf{aut}_\circ$ is the constructor of the automaton state. $n \in S$ denotes the current state of the automaton. E contains the values of the automaton attributes.

Given an automaton $a \in \mathsf{Automaton}$, we denote by $a.Field$ with $Field \in \{\Sigma, S, \nu, \delta, SF, DF, n_0\}$ the corresponding component of the tuple, i.e., $a.n_0$ denotes the initial state of a.

Functions *init* and *final* are now defined as follows. Let a be an automaton ASTD.

$$\mathit{init}(a) \triangleq (\mathsf{aut}_\circ, a.n_0, a.E_{init}, \mathit{init}(a.\nu(n_0)))$$

$$\mathit{final}(a, (\mathsf{aut}_\circ, n, E, s)) \triangleq n \in a.SF \vee (n \in a.DF \wedge \mathit{final}(a.\nu(n), s))$$

Symbol E_{init} denotes the initial values of attributes, as specified in their declaration. $\mathit{init}(a.\nu(n_0))$ returns the initial state of the sub-ASTD $\nu(n_0)$ of n_0. For example,

in Fig. 1, $init(A.\nu(1)) = \mathsf{elem}$ as state 1 is elementary and $init(\mathsf{A}) = (\mathsf{aut}_\circ, 1, \{(x,0)\}, \mathsf{elem})$. A deep final state is final only when its sub-ASTD is also final, whereas a shallow final state is final irrespective of the state of its sub-ASTD.

2) Semantics: There are six rules of inference to define the semantics of an automaton, in order to deal with the different types of transitions and states. The two most frequently used rules are illustrated here. The other rules are defined in [20].

The first rule, aut_1, describes a transition between local states.

$$\mathsf{aut}_1 \ \frac{a.\delta((\mathsf{loc}, n_1, n_2), \sigma', g, A_{tr}, final?) \quad \Psi \quad \Omega_{loc}}{(\mathsf{aut}_\circ, n_1, E, s) \xrightarrow{\sigma, E_e, E'_e}_a (\mathsf{aut}_\circ, n_2, E', init(a.\nu(n_2)))}$$

The conclusion of this rule states that a transition on event σ can occur from n_1 to n_2 with before and after automaton attributes values E, E'. The state of the sub-ASTD of n_2 is its initial state (i.e., $init(a.\nu(n_2))$). The premiss provides that such a transition is possible if there is a matching transition in δ, which is represented by $\delta((\mathsf{loc}, n_1, n_2), \sigma', g, A_{tr}, final?)$. σ' is the event labelling the transition, and it may contain variables. The value of these variables is given by the environment E_e and local attributes values E, which can be applied as a substitution to a formula using operator $(\![\ ]\!)$. This match on the transition is provided by premiss Ψ defined as follows.

$$\Psi \triangleq \big((final? \Rightarrow final(a, (\mathsf{aut}_\circ, n_1, E, s))) \ \wedge \ g \ \wedge \ \sigma' = \sigma\big)(\!|E_g|\!)$$

Ψ can be understood as follows. If the transition is final (i.e., $final? = \mathsf{true}$), then the current state must be final. The transition guard g holds. The event received, noted σ, is equal to the event σ' which labels the automaton transition, after applying the environment E_g as a substitution. Environment E_g is defined in premiss Ω_{loc}.

$$\Omega_{loc} \triangleq \left\{ \begin{array}{c} A = A_{tr} \ ; \ a.A_{astd} \\ E_g = E_e \lhd\!\!\lhd E \\ A(E_g, E'_g) \\ E'_e = E_e \lhd\!\!\lhd (V \lhd E'_g) \\ E' = V \lhd E'_g \end{array} \right\}$$

Premiss Ω_{loc} uses the relational domain restriction operator $U \lhd r = \{x \mapsto x' \mid x \in U \wedge x \mapsto x' \in r\}$, where r is a relation and U a set, and the domain subtraction $U \lhd\!\!\lhd r = \{x \mapsto x' \mid x' \notin U \wedge x \mapsto x' \in r\}$, and the override $r_1 \lhd\!\!\lhd r_2 = (dom(r_2) \lhd\!\!\lhd r_1) \cup r_2$. The execution of an action A on attributes E with possible after value E' is noted $A(E, E')$. The sequential execution of actions A_1 and A_2 is noted $A_1 \ ; \ A_2$. Premiss Ω_{loc} can be understood as follows. The actions executed are the transition action A_{tr}, followed by the ASTD action $a.A_{astd}$, which is declared in the heading of the automaton. The ASTD action is useful to factor out state modifications that must be done on every transition of the ASTD. Symbol E_g, defined as $E_e \lhd\!\!\lhd E$, denotes the global list of variables that can be modified by the actions. It includes the variables declared in the enclosing ASTDs (E_e) and the variables declared

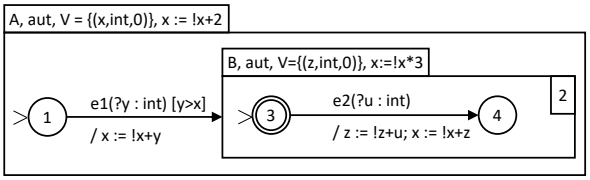


Fig. 1: An automaton ASTD with a complex state 2

locally (E). When a local variable bears the same name as a variable declared in the enclosing ASTDs, it overrides it, similarly to shadowing in programming languages like C, which is represented using the override operator ($\lhd\!\!\lhd$). Their after values E'_g are used to set E' (the local attributes) using the restriction on the attributes V declared in the ASTD and the values E'_e (the attributes declared in enclosing ASTDs), to model variable shadowing.

Rule aut_6, handles transitions within the sub-ASTD $a.\nu(n)$ of state n.

$$\mathsf{aut}_6 \ \frac{s \xrightarrow{\sigma, E_g, E''_g}_{a.\nu(n)} s' \qquad \Theta}{(\mathsf{aut}_\circ, n, E, s) \xrightarrow{\sigma, E_e, E'_e}_a (\mathsf{aut}_\circ, n, E', s')}$$

$$\Theta \triangleq \left(\begin{array}{cc} E_g = E_e \lhd\!\!\lhd E & a.A_{astd}(E''_g, E'_g) \\ E'_e = E_e \lhd\!\!\lhd (V \lhd E'_g) & E' = V \lhd E'_g \end{array} \right)$$

The transition starts from a sub-state s and moves to the sub-state s' of state n. Actions are executed bottom-up. E''_g denotes the values computed by the sub-ASTD. Premiss Θ defines the computation of E'_g from E''_g by executing the ASTD action A_{astd}. E'_e and E' are extracted by partitioning E'_g using V. Premiss Θ is reused in all subsequent rules where a sub-ASTD transition is involved.

3) Example: Fig. 1 provides an automaton ASTD that we can use to illustrate transitions and transition execution proofs. Automaton A declares an attribute set V={(x,int,0)} which contains attribute x, of type int with initial value 0. Automaton A also declares an action x:=!x+2. Actions are expressed in OCaml; expression !x denotes the before value of x. State 1 is an elementary state depicted by ⟩○. State 2 of ASTD A is a complex state, the automaton ASTD B. The transition from 1 to 2 is labeled with event e1(?y : int), which declares a local variable y whose scope is only the transition. It also contains guard [y > x] and an action x:=!x+y. ASTD B, of state 2, declares a local variable z and an ASTD action x := !x∗3. Its initial state is also final and it is computed by the function $init(\nu(2)) = (\mathsf{aut}_\circ, 3, \{(z,0)\}, \mathsf{elem})$.

ASTD A can execute the following two transitions.

$$\begin{array}{ll} & (\mathsf{aut}_\circ, 1, \{(x,0)\}, \mathsf{elem}) \\ \xrightarrow{e1(1)}_A & (\mathsf{aut}_\circ, 2, \{(x,3)\}, (\mathsf{aut}_\circ, 3, \{(z,0)\}, \mathsf{elem})) \\ \xrightarrow{e2(1)}_A & (\mathsf{aut}_\circ, 2, \{(x,14)\}, (\mathsf{aut}_\circ, 4, \{(z,1)\}, \mathsf{elem})) \end{array}$$

The proof of the first transition is the following, stripping the keywords $\mathsf{aut}_\circ$ and elem for the sake of concision.

$$\mathsf{env}\ \dfrac{\mathsf{aut}_1\ \dfrac{\delta((\mathsf{loc},1,2),e1(y),y>x,x:=!x+y,\mathsf{false})\quad (y>x\wedge e1(y)=e1(1))([x:=0])}{(1,\{(x,0)\})\xrightarrow{e1(1),\{\},\{\}}_A (2,\{(x,3)\},(3,\{(z,0)\}))}}{(1,\{(x,0)\})\xrightarrow{e1(1)}_A (2,\{(x,3)\},(3,\{(z,0)\}))}$$

Rule env adds the empty environments. Rule aut_1 succeeds, because y is valued to 1 by the equality $e1(y) = e1(1)$ and the guard $y > x$ holds after substituting x with 0. The symbols of premiss Ω_{loc} in step aut_1 are valued as follows.

$$\begin{aligned}
&E = \{(x,0)\} \quad E_e = \{\}\\
&E_g = \{\} \lhd\!\!\!\cdot\ \{(x,0)\} = \{(x,0)\}\\
&E'_g = \{(x,3)\},\\
&\quad \text{since}\{\mathsf{x}{:=}!\mathsf{x}{+}1;\ \mathsf{x} := !\mathsf{x}{+}2\}(\{(x,0)\},\{(x,3)\})\\
&E'_e = \{\} \lhd\!\!\!\cdot\ (\{x\} \lhd\!\cdot\ \{(x,3)\}) = \{\}\\
&E' = \{x\} \lhd\!\cdot\ \{(x,3)\} = \{(x,3)\}
\end{aligned}$$

The proof of the second transition is the following.

$$\mathsf{env}\ \dfrac{\mathsf{aut}_6\ \dfrac{\mathsf{aut}_1\ \dfrac{\delta((\mathsf{loc},3,4),e2(u),\mathsf{true},z:=!z+u;x:=!x+z,\mathsf{false})\quad (e2(u)=e2(1))([x:=3,z:=0])}{(3,\{(z,0)\})\xrightarrow{e2(1),\{(x,3)\},\{(x,12)\}}_B (4,\{(z,1)\})}}{(2,\{(x,3)\},(3,\{(z,0)\}))\xrightarrow{e2(1),\{\},\{\}}_A (2,\{(x,14)\},(4,\{(z,1)\}))}}{(2,\{(x,3)\},(3,\{(z,0)\}))\xrightarrow{e2(1)}_A (2,\{(x,14)\},(4,\{(z,1)\}))}$$

B. Sequence

The sequence ASTD allows for the sequential composition of two ASTDs. When the first item reaches a final state, the second one can start its execution. This enables decomposing problems into a set of tasks that have to be executed in sequence.

1) Syntax: A sequence ASTD is a structure $\langle \fatsemi, fst, snd\rangle$ where fst, snd are ASTDs denoting respectively the first and second sub-ASTDs of the sequence. A sequence state is of type $\langle \fatsemi_\circ, E, [\mathsf{fst} \mid \mathsf{snd}], s\rangle$, where $\fatsemi_\circ$ is a constructor of the sequence state, E the values of attributes declared in the sequence, $[\mathsf{fst} \mid \mathsf{snd}]$ is a choice between two markers that respectively indicate whether the sequence is in the first sub-ASTD or the second sub-ASTD and $s \in \mathsf{States}$. Functions *init* and *final* are defined as follows. Let a be a sequence ASTD.

$$\begin{aligned}
\mathit{init}(a) &\triangleq (\fatsemi_\circ, a.E_{init}, \mathsf{fst}, \mathit{init}(a.fst))\\
\mathit{final}(a,(\fatsemi_\circ,E,\mathsf{fst},s)) &\triangleq \mathit{final}(a.fst,s) \wedge \mathit{final}(a.snd,\mathit{init}(a.snd))\\
\mathit{final}(a,(\fatsemi_\circ,E,\mathsf{snd},s)) &\triangleq \mathit{final}(a.snd,s)
\end{aligned}$$

The initial state of a sequence is the initial state of its first sub-ASTD. A sequence state is final when either i) it is executing its first sub-ASTD and this one is in a final state, and the initial state of the second sub-ASTD is also a final state; ii) it is executing the second sub-ASTD which is in a final state.

2) Semantics: Three rules are necessary to define the execution of the sequence. Rule $\fatsemi_1$ deals with transitions on the sub-ASTD *fst* only. Rule $\fatsemi_2$ deals with transitions from *fst* to *snd*, when *fst* is in a final state. Rule $\fatsemi_3$ deals with transitions on the sub-ASTD *snd*.

$$\fatsemi_1\ \frac{s \xrightarrow{\sigma,E_g,E''_g}_{a.fst} s' \qquad \Theta}{(\fatsemi_\circ,E,\mathsf{fst},s)\xrightarrow{\sigma,E_e,E'_e}_a (\fatsemi_\circ,E',\mathsf{fst},s')}$$

$$\fatsemi_2\ \frac{\mathit{final}(a.fst,s)([E_g]) \quad \mathit{init}(a.snd)\xrightarrow{\sigma,E_g,E''_g}_{a.snd} s' \quad \Theta}{(\fatsemi_\circ,E,\mathsf{fst},s)\xrightarrow{\sigma,E_e,E'_e}_a (\fatsemi_\circ,E',\mathsf{snd},s')}$$

$$\fatsemi_3\ \frac{s \xrightarrow{\sigma,E_g,E''_g}_{a.snd} s' \qquad \Theta}{(\fatsemi_\circ,E,\mathsf{snd},s)\xrightarrow{\sigma,E_e,E'_e}_a (\fatsemi_\circ,E',\mathsf{snd},s')}$$

C. Parameterized Synchronization

Synchronization ASTDs allow managing concurrent resources between two ASTD components using the synchronization operator $|[]|$. Events and variables between the two components are recorded in a synchronization set Δ for handling synchronization.

1) Syntax: A parameterized synchronization ASTD is a structure $\langle |[]|, \Delta, l, r\rangle$ where Δ is the synchronization set of event labels, $l, r \in \mathsf{ASTD}$ are the synchronized ASTDs. When the label of the event belongs to Δ, the two sub-ASTDs must both execute it; otherwise either the left or the right sub-ASTD can execute it; if both sub-ASTDs can execute it, the choice between them is nondeterministic. When $\Delta = \varnothing$, the synchronization is called an interleaving, noted $|||$.

A parameterized synchronization state is of type $\langle |[]|_\circ, E, s_l, s_r\rangle$, where s_l, s_r are the states of the left and right sub-ASTDs. Initial and final states are defined as follows. Let a be a parameterized synchronized ASTD.

$$\begin{aligned}
\mathit{init}(a) &\triangleq (|[]|_\circ, a.E_{init}, \mathit{init}(a.l), \mathit{init}(a.r))\\
\mathit{final}(a,(|[]|_\circ,E,s_l,s_r)) &\triangleq \mathit{final}(a.l,s_l) \wedge \mathit{final}(a.r,s_r)
\end{aligned}$$

2) Semantics: There are three inference rules. Rules $|[]|_1$ and $|[]|_2$ respectively describe execution of events, with no synchronization required, either on the left or the right sub-ASTDs. Rule $|[]|_1$ below caters for execution on the left sub-ASTD. The function $\alpha(e)$ returns the label of event e.

$$|[]|_1\ \frac{\alpha(\sigma)\notin\Delta \qquad s_l \xrightarrow{\sigma,E_g,E''_g}_{a.l} s'_l \qquad \Theta}{(|[]|_\circ,E,s_l,s_r)\xrightarrow{\sigma,E_e,E'_e}_a (|[]|_\circ,E',s'_l,s_r)}$$

Rule $|[]|_2$ is symmetric to $|[]|_1$ and indicates behaviour when the right side execute the action.

$$|[]|_2\ \frac{\alpha(\sigma)\notin\Delta \qquad s_r \xrightarrow{\sigma,E_g,E''_g}_{a.r} s'_r \qquad \Theta}{(|[]|_\circ,E,s_l,s_r)\xrightarrow{\sigma,E_e,E'_e}_a (|[]|_\circ,E',s_l,s'_r)}$$

The most interesting case is when the left and right sub-ASTDs must synchronize on an event (i.e., when $\alpha(\sigma) \in$

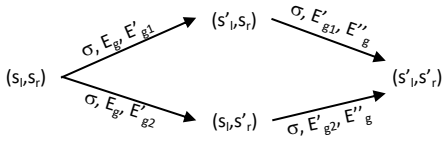


Fig. 2: Commutativity of actions execution in a parameterized synchronization on σ

Δ). Consider the transitions on the left and right sub-ASTDs when each of them is executed independently of the other.

$$\Omega_{ilr} \triangleq \left(s_l \xrightarrow{\sigma,E_g,E'_{gl}}_{a.l} s'_l \qquad s_r \xrightarrow{\sigma,E_g,E'_{gr}}_{a.r} s'_r \right)$$

Since shared variables (E_g) can be modified by both sub-ASTDs, their modifications could be inconsistent, which should forbid the synchronization transition. This requires to check that $E'_{gl} = E'_{gr}$. However, when one variable is modified by both sub-ASTDs, the natural intent is typically that the compound result of both sides is desired, that is, to assume that one side executes on the values returned by the other. For instance, assume that both sub-ASTDs increment shared variable x by 1 and assume that the before value of x is 0. The above semantics gives $x = 1$, whereas the compound result is $x = 2$. If one sub-ASTD increments by 1 and the other by 2, the above semantics forbids execution because the result is inconsistent, whereas the compound execution returns 3. Executing the left sub-ASTD before the right sub-ASTD is specified as follows.

$$\Omega_{lr} \triangleq \left(s_l \xrightarrow{\sigma,E_g,E'_{g1}}_{a.l} s'_l \qquad s_r \xrightarrow{\sigma,E'_{g1},E''_g}_{a.r} s'_r \right)$$

Executing the right sub-ASTD before the left sub-ASTD is specified as follows.

$$\Omega_{rl} \triangleq \left(s_r \xrightarrow{\sigma,E_g,E'_{g2}}_{a.r} s'_r \qquad s_l \xrightarrow{\sigma,E'_{g2},E''_g}_{a.l} s'_l \right)$$

Since synchronization is commutative, i.e., both execution orders should return the same states. Fig. 2 illustrates this property. Checking this commutativity at each transition is expensive. To improve performance, it could be statically checked using proof obligations, or by analysis of the variables read and written by each sub-ASTD, to ensure that the left and right sub-ASTDs are independent (i.e., they do not modify the same variables and one sub-ASTD does not modify the variables read by the other). When the synchronization operands are nondeterministic, this is still expensive to execute, because a commutative combination must be found, which means enumerating combinations of possibilities from both sides. Still, this is our preferred semantics for a synchronization, because it works well for deterministic specifications. Here is the rule for synchronization.

$$|[]|_3 \frac{\alpha(\sigma) \in \Delta \qquad \Omega_{lr} \qquad \Omega_{rl} \qquad \Theta}{(|[]|_\circ, E, s_l, s_r) \xrightarrow{\sigma,E_e,E'_e}_a (|[]|_\circ, E', s'_{l_1}, s'_{r_1})}$$

Note that our treatment of synchronization differs from those in CSP, CSP‖B, CSP2B or Circus. Since CSP is a pure process algebra, it does not support attributes. In CSP‖B and CSP2B, a single operation is executed to update a set of global state variables; the problem of having two different actions modifying the same variable does not exist. However, if the action to execute depends on the state of the CSP expression, extra state variables must be added to encode the CSP state in the B machine, which introduces some undesirable coupling between the CSP specification and the B specification, breaking the ideal separation between the control part represented in CSP and the data part represented in B. In Circus, synchronization occurs on channels only; state variables are not modified on a synchronization between two processes; actions are executed in interleave. The ASTD action is particularly useful for the synchronization ASTD, because it enables to make decisions and updates based on the result of the execution in both sub-ASTDs.

Also note that to process and combine action effects performed in sequence or in parallel, there are multiple alternatives available. The B [2] style parallel updates are quite restrictive, and disallow parallel assignments to the same variables. The ASM approach [1] is to collect parallel updates and perform them at the end of an (atomic) event. The translation from ASM to B in [21] proposes to use update functions which are composed. While elegant, the actions in our ASTD implementation take effect immediately and are executed by a "black box" interpreter. Hence this was not a practical approach.

D. Flow

It is quite common that the same event e can be part of several cyber attack specifications, as illustrated in Fig. 3. An intrusion detection system needs to execute such an event on each attack specification that can execute it; this behaviour is not fulfilled by either interleaving or synchronization of attack specifications. This raises the need of a new operator $\Cup$, called flow, which will execute the event on each sub-ASTD whenever possible, like AND states in statecharts. In contrast to other ASTD operators, the rules for this operator involve negation, i.e., one has to determine whether an event is not possible for sub-ASTDs. In that respect it is related to FDR's priority annotation for CSP [22]. Such an annotation could probably be used to implement $\Cup$.

In Fig. 3, Attack0 combines multiple attacks using a flow: Attack1, Attack2 and Attack3. The flow operator checks all possible transitions from each attack component and executes them. When event e is received, transitions 0-1, 3-4 and 7-8 are executed if their ASTDs are in state 0, 3 and 7, respectively. Other transitions (i.e., 1-2, 4-5 and 6-7) execute in interleaving.

1) Syntax: A flow ASTD is a structure $\langle \Cup, l, r \rangle$. A flow state is of type $\langle \Cup_\circ, E, s_l, s_r \rangle$, where s_l, s_r are the states of the left and right sub-ASTDs. Initial and final states are

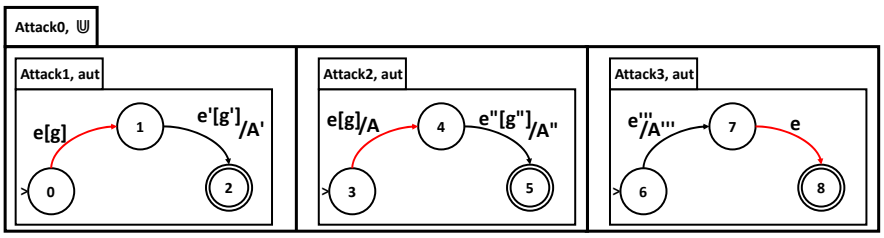


Fig. 3: Using the Flow operator to combine multiple attack models

defined as follows. Let a be a flow ASTD.

$$\begin{aligned} \mathit{init}(a) &\triangleq (\Psi_\circ, a.E_{init}, \mathit{init}(a.l), \mathit{init}(a.r)) \\ \mathit{final}(a, (\Psi_\circ, E, s_l, s_r)) &\triangleq \mathit{final}(a.l, s_l) \wedge \mathit{final}(a.r, s_r) \end{aligned}$$

2) Semantics: We use the following abbreviation to denote that an ASTD cannot execute a transition from a state s and global attributes E_g.

$$s \xrightarrow{\sigma, E_g}_a \bot \triangleq \neg \exists E'_g, s' \cdot s \xrightarrow{\sigma, E_g, E'_g}_a s'$$

Symbol $\bot$ is used to denote an undefined value. The negation of a transition predicate is computed using the usual negation as failure approach. For example (Fig. 3), the execution of transitions 0 to 1 and 3 to 4 fails on the reception of the event e'''. Then, they are ignored while the transition 6 to 7 is executed. Here are the first two rules when only one of the two sub-ASTDs can execute the event.

$$\Psi_1 \frac{s_l \xrightarrow{\sigma, E_g, E''_g}_{a.l} s'_l \quad s_r \xrightarrow{\sigma, E''_g}_{a.r} \bot \quad \Omega_{lr} \Leftrightarrow \Omega_{rl} \quad \Theta}{(\Psi_\circ, E, s_l, s_r) \xrightarrow{\sigma, E_e, E'_e}_a (\Psi_\circ, E', s'_l, s_r)}$$

$$\Psi_2 \frac{s_r \xrightarrow{\sigma, E_g, E''_g}_{a.r} s'_r \quad s_l \xrightarrow{\sigma, E''_g}_{a.l} \bot \quad \Omega_{lr} \Leftrightarrow \Omega_{rl} \quad \Theta}{(\Psi_\circ, E, s_l, s_r) \xrightarrow{\sigma, E_e, E'_e}_a (\Psi_\circ, E', s_l, s'_r)}$$

The premiss $\Omega_{lr} \Leftrightarrow \Omega_{rl}$ ensures that one execution order succeeds iff the other also succeeds. It ensures the determinacy of the flow operator.

The third rule describes the case where both sub-ASTDs can execute the event; it is almost the same as $|[]|_3$, as it requires commutativity.

$$\Psi_3 \frac{\Omega_{lr} \quad \Omega_{rl} \quad \Theta}{(\Psi_\circ, E, s_l, s_r) \xrightarrow{\sigma, E_e, E'_e}_a (\Psi_\circ, E', s'_{l_1}, s'_{r_1})}$$

E. Quantified Synchronization

The quantified synchronization allows for the modeling of an arbitrary number of instances of an ASTD which are executing in parallel, synchronizing on events from Δ.

1) Syntax: A quantified synchronization ASTD is a structure $\langle |[]|:, x, T, \Delta, b \rangle$ where $x \in \mathsf{Var}$ a quantified variable that can be only accessed in read-only mode, T the type of x, $\Delta \subseteq \mathsf{Label}$ a synchronization set of event labels and $b \in \mathsf{ASTD}$ the body of the synchronization. The state of a quantified synchronization is of type $\langle |[]|:_\circ, E, f \rangle$ where $|[]|:_\circ$ is the constructor, E the values of attributes and $f \in T \to \mathsf{States}$ is a function which associates a state of b to each value of T. Initial and final states are defined as follows. Let a be a quantified synchronized ASTD.

$$\begin{aligned} \mathit{init}(a) &\triangleq (|[]|:_\circ, a.E_{init}, T \times \{\mathit{init}(a.b)\}) \\ \mathit{final}(a, (|[]|:_\circ, E, f)) &\triangleq \forall c : T \cdot \mathit{final}(a.b, f(c)) \end{aligned}$$

2) Semantics: Rule $|[]|:_1$ describe execution of events with no synchronization. Symbol c denotes the element of T chosen for the execution.

$$|[]|:_1 \frac{\alpha(\sigma) \notin \Delta \quad f(c) \xrightarrow{\sigma, E_g \Lsh \{x \mapsto c\}, E''_g}_{a.b} s' \quad \Theta}{(|[]|:_\circ, E, f) \xrightarrow{\sigma, E_e, E'_e}_a (|[]|:_\circ, E', f \Lsh \{c \mapsto s'\})}$$

Rule $|[]|:_2$ describe execution of an event with synchronization. All elements of T must execute σ, in any order. It generalizes rule $|[]|_3$ and requires commutativity.

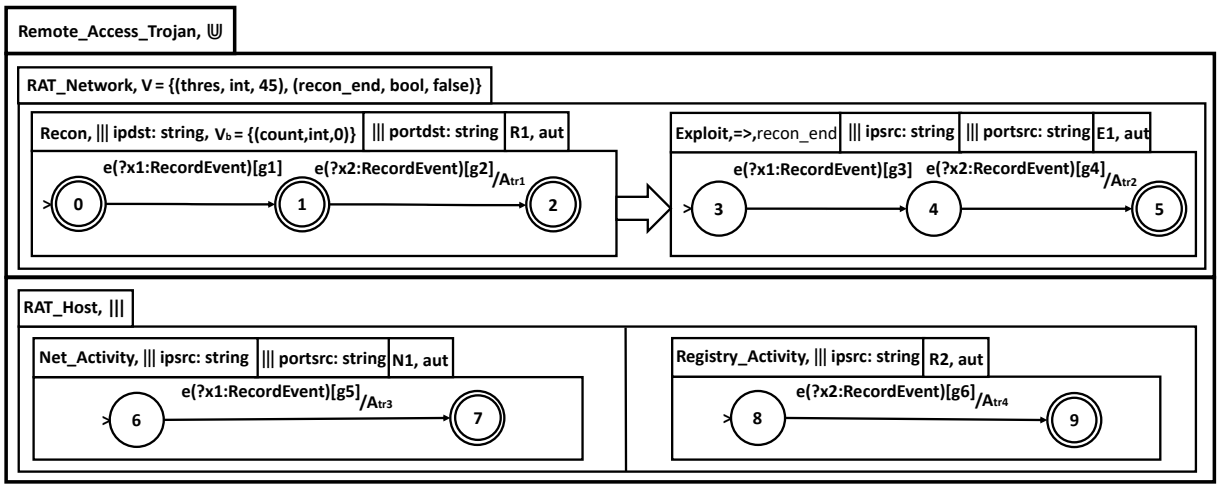
$$|[]|:_2 \frac{\alpha(\sigma) \in \Delta \quad \Omega_{qsyn} \quad \Theta}{(|[]|:_\circ, E, f) \xrightarrow{\sigma, E_e, E'_e}_a (|[]|:_\circ, E', f')}$$

Premiss Ω_{qsyn} formalizes commutativity by universally quantifying over all permutations p of T (noted $p \in \pi(T)$) and using Es as a sequence of environments storing the intermediate results of the computation of E''_g from E_g by iterating over the elements $p(i)$ of p. Let $k = |T|$.

$$\Omega_{qsyn} \triangleq \left(\begin{array}{l} \forall p \in \pi(T) \cdot \exists Es \in 0..k \to \mathsf{Env} \wedge Es(0) = E_g \\ \wedge\ E(k) = E''_g \wedge \forall i \in 1..k \cdot (\\ f(p(i)) \xrightarrow{\sigma, Es(i-1) \Lsh \{x \mapsto p(i)\}, Es(i)}_{a.b} f'(p(i))) \end{array} \right)$$

III. Case Study

Fig. 4 illustrates a case study of ASTDs in cyber attack detection. Increasingly, attackers develop various strategies to break down existing defence systems and, consequently, gain unauthorized access to a private Information System (IS). They operate strategically by executing a sequence of threatening actions [23] to discover the network topology and vulnerabilities on the target IS, followed by a phase exploiting the vulnerabilities found to command and control the target. We illustrate this approach by an active attack called Remote Access Trojan (RAT) [24]. A RAT attack operates both on a network and a host. It starts on a network through vulnerability scans. Next, the attacker sends a malware to the victim machine (e.g., using email spams) for exploitation. Once the malware is installed on the victim's machine, it tries to automatically connect to the attacker machine, and an attack session is opened



g1 = (x1.ipdst=ipdst) && (x1.portdst=portdst) && (x1.proto="TCP") && (x1.tcpflags= "S") && (**not** recon_end)
g2 = (x2.ipsrc=ipdst) && (x2.portsrc=portdst) && (x2.proto="TCP") && (x2.tcpflags= "RA")
g3 = (x1.ipsrc=ipsrc) && (x1.portsrc=portsrc) && (x1.proto="TCP") && (x1.tcpflags= "S")
g4 = (x2.ipdst=ipsrc) && (x2.portdst=portsrc) && (x2.proto="TCP") && (x2.tcpflags= "SA") && (contains x2.payload "stdapi")
g5 = (x1.ipsrc=ipsrc) && (x1.portsrc=portsrc) && (x1.proto="TCP") && (x1.eventid= "3")
&& (**List.mem** x1.portdst ["80"; "443"; "444"; "445"; "4444"]))
g6 = (x2.ipsrc=ipsrc) && (x2.eventid= "13") && (contains x2.registrykey "HK\\(U\\|LV\\|LM\\|CR\\)\\\\") &&
(contains x2.registrykey "\\\\Run\\\\[a-zA-Z]+ .\\(exe\\|vbs\\|bat\\|lnk\\)")

Atr1 = { count := !count+1;
if !count >= !thres **then**
alert "Port scan attack";
count := 0; recon_end := **true**;;}

Atr2 = {alert "Metasploit privilege escalation";;}

Atr3 = {alert "File ' "^x1.image^" ' attempted a suspicious connection to "^x1.ipdst^" on port "
^x1.portdst;;}

Atr4 = {alert "Malware registry attack";;}

Fig. 4: Remote Access Trojan ASTD specification

when user clicks on the infected program (exploitation). Then, events are generated from both host and network sides. Considering both gives better insights (or a holistic view) of RAT activities.

The main ASTD is identified by the attack name Remote_Access_Trojan in the tab of the box. The name can be omitted for nested ASTDs. Remote_Access_Trojan illustrates the extension: it declares two attributes, thres and recon_end, with their types and initial values. Remote_Access_Trojan is a flow ASTD (denoted by ⋓) that concurrently executes events from two attack models in a network and host: RAT_Network and RAT_Host. RAT_Network allows for sequential composition of two ASTDs: Recon and Exploit. The first ASTD of the sequence (i.e., Recon) must reach a final state before the next one can start.

The ASTD Recon starts its execution and inspects the network traffic to detect port scanning and operating system (OS) detection attempts. Attempts may be done by an attacker who tries to scan open ports and OS vulnerabilities (e.g., system errors, bugs) on the target IS (victim). Attacker actions generate network traffic that contains malicious patterns enabling identifying the current attack. Recon is a quantified interleaving ASTD (denoted by ||| ipdst : string) that allows an arbitrary number of instances of the nested ASTD to be executed in interleaving, each instance being indexed by its ip address ipdst. It also declares an attribute count whose value is shared by all its interleave instances. Note that attributes can be declared within any ASTD. ASTD Recon has a nested nameless ASTD which is a quantified interleaving on the destination port (i.e., ||| portdst : string). For each possible value taken by ipdst and portdst, the nested automaton R1 tracks the scanning of a port; it also has read-write access of the attributes count and thres. However, R1 can not modify ipdst and portdst, because quantified interleave variables are read-only. Being called within two quantified interleaves, there is an instance of this automaton for each pair of values of ipdst and portdst. Its initial state is 0, depicted by ⟩○ and it is also final (denoted by ◎). This state has an outgoing transition labeled by the event e(?x1:RecordEvent) and a guard [g1]. Variable x1 is a local variable whose scope is the transition only. Type RecordEvent is a record containing both host and network events in the simplified form

$$\langle systime, eventid, image, registrykey, proto, ipsrc, portsrc, ipdst, portdst, tcpflags, payload\rangle$$

where *systime* is the date-time when the event occurred, *eventid* is the event ID [25] (e.g., "1"=Process Creation, "3"=Network Connection, "13"=Registry Value Set), *image* the file path of the running process, *registrykey* the path of the registry object, *proto* is the protocol

type (e.g., ICMP, TCP, UDP), *ipsrc* the source address, *portsrc* the source port, *ipdst* the destination address, *portdst* the destination port, *flags* a combination of TCP flags (S/SYN initiates a connection, A/ACK acknowledges received data, R/RST aborts a connection in response to ans error), and *payload* the event content.

The guard states that the source initiates a TCP connection to the destination, the destination ip address and port of the event matches the quantified interleave variables ipdst and portdst, and the end of the reconnaissance phase has not been reached. This means that the event is executed only on the appropriate automaton instance. The transition from state 1 to state 2 captures the response of the victim to the attacker. It illustrates the declaration of actions on transitions, the second type of extensions made to the ASTD notation. Action $\mathsf{A}_{\mathsf{tr1}}$ increases attribute count of Recon. When count reaches the threshold, attribute recon_end is set to true, which will enable ASTD Exploit to start. Actions are expressed in OCaml to integrate easily with the ASTD interpreter. An attribute x is represented by an OCaml reference variable x, which is dereferenced using "$!x$" to access its value.

Let consider the following events which are used to describe the RAT specification.

```
revt1 <systime="t1", eventid="", image="", registrykey="", proto="TCP",
ipsrc="ip1", portsrc="pt1", ipdst="ip2", portdst="pt2", tcpflags="S",
payload="p1">

revt2 <systime="t2", eventid="", image="", registrykey="", proto="TCP",
ipsrc="ip2", portsrc="pt2", ipdst="ip1", portdst="pt1", tcpflags="RA",
payload="p2">

revt3 <systime="t3", eventid="", image="", registrykey="", proto="TCP",
ipsrc="ip1", portsrc="pt1", ipdst="ip2", portdst="pt3", tcpflags="S",
payload="p3">

revt4 <systime="t4", eventid="", image="", registrykey="", proto="TCP",
ipsrc="ip2", portsrc="pt3", ipdst="ip1", portdst="pt1", tcpflags="RA",
payload="p4">

revt5 <systime="t5",eventid="3",image="C:\\Users\\admin\\KWIAYAMw\\
XYkgwUUo.exe", registrykey="", proto="TCP", ipsrc="ip2",portsrc="pt2",
ipdst="ip1", portdst="pt1",tcpflags="S",payload="p5">

revt6 <systime="t6", eventid="", image="", registrykey="", proto="TCP",
ipsrc="ip1", portsrc="pt1", ipdst="ip2", portdst="pt2", tcpflags="SA",
payload="stdapi">

revt7 <systime="t7",eventid="13",image="C:\\ProgramData\\pykEMEsI\\
ykAYMkMA.exe",registrykey="HKLM\\SOFTWARE\\Microsoft\\Windows\\
CurrentVersion\\Run\\EUgUgAYs.exe", proto="UDP",ipsrc="ip1",
portsrc="pt4",ipdst="ip1",portdst="pt5",tcpflags="",payload="">
```

An attacker initiates a TCP connection on its machine (address ip1) to the victim machine (address ip2). The reception of the event e(revt1) triggers a transition from 0 to 1, within Recon in the interleavings instance ip2 and pt2. The state 2 is reached when event revt2 is received. Action $\mathsf{A}_{\mathsf{tr1}}$ increments attribute count and returns an immediate alert when the number of scanned ports on the victim machine reaches a threshold. An alert is a message sent to the environment (e.g., stdout). In this case, the attribute recon_end takes the value true, enabling the next component of the sequence (i.e., Exploit) to start. After receiving event revt4, the value of count is 2 and two instances of R1 (i.e., (ip2,pt2) and (ip2,pt3) are in state 2; the others are still in their initial state).

Because ASTD Recon is a quantified interleave, it is final when all its interleaved instances are final; similarly for its nested ASTD ||| portdst : string. An automaton is final when its current state is final. Since all the states of R1 are final, then ASTD Recon is always final. Thus, ASTD Exploit is always enabled, but it is a guard ASTD, identified by the operator ⇒. It can start only if its guard condition (recon_end) is true. The introduction of attributes allows us to define more specific guard conditions that can be used in guarded ASTDs in a sequential composition. It is an alternative way of controlling the sequential composition of ASTDs: since each automaton state is final, it is the guard that decides when the second component of a sequence can trigger.

The exploit phase starts when the victim machine runs an infected program (weapon) that communicates with the attacker machine. It allows the attacker to send malicious payloads to the victim by exploiting the Server Message Block (SMB) vulnerability [26]. Payloads contain standard remote call API (STDAPI) signatures that enable detecting attacker activities. The nested ASTDs of Exploit respectively interleave on the source address (i.e., ||| ipsrc : string) and the destination address (i.e., ||| ipdst : string). This allows a number of instances of E1 for each possible of ipsrc and ipdst. The automaton ASTD E1 is in state 3.

On the reception of an event e(revt5), transitions from 3 to 4 and 6 to 7 are synchronously executed by the flow operator. It means that the victim has clicked on the malware file and exploit succeeded. In the host, the malware attempts a network connection (i.e., revt5.eventid="3") to the attacker machine on port 4444. The function List.mem elt list returns true when elt exists in list. In the network, an attack session has been initiated (i.e., revt5.flags="S") between the victim and the attacker. The next transition (i.e., from 4 to 5) is executed when an attack session is established (i.e., revt6.flags="SA") and the attacker starts to send malicious payloads (e.g., "stdapi" in the revt6 payload). Concurrently, the malware attempts to locally modify registries (i.e., revt7.eventid="13") on the victim machine ip1. Its behaviour is identified by a regular expression in the registry key [27]. The function contains str regexp returns true, when a string in str matches the regular expression regexp.

IV. ASTD Tool Support

A. Prolog and ProB

The ProB model checker was originally developed for B specifications, but can also be applied to other languages whose operational semantics are expressed in Prolog. This is how the synchronization between CSP and B was realized in [28], by transcribing the operational semantics of CSP to Prolog. We have implemented the operational semantics of extended ASTDs along with all features and operators as required for this case study: sequence, guard, and various synchronization operators (interleaving, full and selective synchronization) and the new flow operator.

The operational semantics rules of extended ASTDs can be translated to individual Prolog clauses. For example, the rule for the sequence operator is translated as follows.

```
atrans(seq([A1|T]),G1,Trans,S2,G2) :- aseq_trans(A1,T,
G1,Trans,S2,G2).
aseq_trans(A1,T,G1,Trans,S2,G2) :- %allow A1 to evolve
atrans(A1,G1,Trans,A2,G2), create_seq(A2,T,S2).
aseq_trans(A1,[A2|T],G1,Trans,S2,G2) :-
% if A1 is final: skip to rest
is_final_astd(A1), aseq_trans(A2,T,G1,Trans,S2,G2).
```

Intuitively, the Prolog predicate `atrans(S1,G1,E,S2,G2)` is true when the ASTD expression S1 can execute the event E in the context of the global state G1, resulting in a new ASTD expression S2 and a new global state G2. In other words, it corresponds to $S1 \xrightarrow{E,G1,G2}_a S2$ from Sect. II. The Prolog predicate `is_final_astd(A)` is true when the ASTD A is final, implementing *final* from Sect. II. For efficiency reasons, `atrans` calls other subsidiary predicates like `aseq_trans` and `create_seq`. The latter simply constructs a new ASTD sequence expression.

An ASTD expression is a Prolog term representing the ASTD structure, e.g., `seq([aut(R1('8.8.8.8',80),aut(E1('8.8.8.8',80)])` to represent the sequential composition of an instance of the R1 automata and an instance of the E1 automata from Fig. 4. The global state is represented as a list of bindings, e.g., `[count/0,recon_end/0,thres/45, warnings/[]]` for R1 in Fig. 4.

Note that here we have implemented sequence not as a binary operator like in Sect. II-B, but also as n-ary operator that combines several ASTDs. The first clause of `aseq_trans` corresponds to the rule $\Rightarrow_1$ from Sect. II-B, and the second clause to rule $\Rightarrow_2$. There is no need for rule $\Rightarrow_3$ as we throw away A1 in the resulting process expression S2.

Currently, to run ProB on an ASTD, one needs to express the transitions of the individual automata in Prolog. For normal automata, this results in one fact per transition. For automata with actions and rules, this corresponds to one Prolog clause per transition, the body of the clause containing the conditions and associated actions. In future, we plan to generate those Prolog translations automatically from the ASTD representation.

By writing the interpreter, we have gained access to various features of ProB: animation, model checking (deadlock, determinism, safety, LTL, CTL), refinement checking, and execution (a faster version of animation which does not store the history of states). In principle one could also synchronize ASTDs with B machines in the style of [28].

As a first simpler example we have replicated the Library case study from [29], using global state to store reservations. Our Prolog ASTD interpreter is about three times faster than ProB on the B translation of the system (not using ProB's symmetry reduction or partial order reduction). On our security case study, the animation features uncovered various issues with earlier versions of the attack models of Fig. 4. For example, the animator and model checker uncovered various unexpected non-determinisms in the attack specification. We have also managed to replay a real attack log, validating that attacks are indeed correctly detected. For this experiment, we have written an ASTD which reads in a log file and replays the events in the file. This ASTD is put into parallel with the attack model, and one can check whether attacks are identified or not. The execution took 50 ms to process 300 events.

B. ASTD OCaml Interpreter

In [30], an initial development of an ASTD interpreter has been made for information systems. It has been extended to support new features for cyber attack detection. The interpreter implements the operational semantics of ASTDs by computing transition proofs. It executes an input attack specification on input sources (i.e., packets, audit logs). Attack specifications are converted into a serialized format [20] before being sent to the interpreter. During execution, raw events from the host or/and network are preprocessed into the enumerated form e(systime, eventid, image, ...). The interpreter reads preprocessed events in offline mode (i.e., from a file) or in real-time, and computes possible transitions of the specification. Actions like alerts are displayed to the administrator.

For the case study, *i*ASTD[1] was able to detect RAT attacks on both the network and the host. The execution took on average 8.621 s to process 89 600 events. Hereafter, some alerts generated from the *i*ASTD output during detection.

```
Alert Port scan attack
Alert Metasploit privilege escalation

Alert File 'C:\\Users\\admin\\Desktop\\ZisUurWz.exe' attempted a suspi
cious connection to 192.168.1.129 on port 4444

Alert File 'C:\\ProgramData\\JAsIssEI\\EUgUgAYs.exe' attempted a suspi
cious connection to 172.217.3.206 on port 80
```

Several attacks like ransomwares and lateral movement have been specified and executed using the interpreter. More complex attacks operating in various environments could also be specified using ASTDs. Being domain-independent, the ASTD language could be extended in a well-defined way to match different operating environments and networks. Thus, *i*ASTD operates as a host-based attack detector, network-based attack detector, and hybrid attack detector when the input ASTD model respectively specifies a host attack behavior, a network attack behavior and both.

V. Discussion and Conclusion

This paper proposes an extension of the ASTD notation with attributes, actions and a new operator called flow. These extensions are particularly useful to model cyber-security attacks and are implemented in two interpreters, one in Prolog with ProB, and the other in OCaml. ProB proved to be a useful addition, because it gives us access to several model checking features already implemented, like refinement checking, determinacy checking, temporal formula checking. However, these interpreters are currently limited to primitive types integers and strings.

[1]The interpreter and results are available at `https://depot.gril.usherbrooke.ca/fram1801/iASTD-public`.

The algebraic nature of our approach is quite useful to explore several variants of specification attacks. For instance, the specification of Fig. 4 triggers the exploit phase detection when one ip address has been scanned. An alternative behaviour is to trigger one exploit phase for each ip address scanned; it suffices to move the sequence operator just after ASTD Recon, and remove the quantified interleave ||| portsrc: string in ASTD Exploit. Another variant would be to use a synchronization ||| instead of a sequence in RAT_Network; this would enable to detect both port scans and exploits concurrently. Making these changes in a model-based language like B, or in a scripting language like Python, which is the common practice in security, would entail several changes in the types of variables and the modification of preconditions and postconditions in several operations, which is subtle and error-prone. The ability to compose specifications relieves the specifier from this burden. The ASTD language is accessible by users who are not necessarily experts and the attack detection is automatically done by the interpreters. A script that reproduces the same behaviour for attack detection, requires a huge case analysis on the content of the packet and audit logs, something which is hard to code and maintain.

Our future work consists in building the ability to define complex types (e.g., Flow, Session, AuditLog) to handle the detection of more complex attacks. We plan to use ontologies to define these types and use them in various tools to support our approach. Further experimentation will be conducted to handle more complex attack specifications. If these are successful, we plan to develop an ASTD compiler that will generate efficient code from ASTD specifications and use them to detect intrusion in real environments. Quantified interleaves can be efficiently executed in constant time or $O(\log(n))$ when the quantification variable occurs in each event, which is typically the case [31].

Acknowledgments. This work was supported in part by NSERC (Natural Sciences and Engineering Research Council of Canada) and CSE (Communications Security Establishment of Canada).

References

[1] E. Börger and R. Stärk, *Abstract state machines: a method for high-level system design and analysis*. Springer Science & Business Media, 2012.

[2] J.-R. Abrial, *The B-book: Assigning Programs to Meanings*. New York, NY, USA: Cambridge University Press, 1996.

[3] J. M. Spivey, *The Z Notation: A Reference Manual*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1989.

[4] C. A. R. Hoare, "Communicating sequential processes," *Commun. ACM*, vol. 21, no. 8, pp. 666–677, Aug. 1978.

[5] A. J. Galloway and W. J. Stoddart, "An operational semantics for zccs," in *First IEEE International Conference on Formal Engineering Methods*, Nov 1997, pp. 272–282.

[6] M. V. M. Oliveira, "Formal derivation of state-rich reactive programs using Circus," Ph.D. dissertation, Department of Computer Science, Univ. of York, 2005.

[7] J. Woodcock and A. Cavalcanti, "A concurrent language for refinement," in *Proceedings of the 5th Irish Conference on Formal Methods*, 2001, pp. 93–115.

[8] S. Schneider and H. Treharne, "CSP theorems for communicating B machines," *Formal Aspects of Computing*, vol. 17, no. 4, pp. 390–422, Dec 2005.

[9] M. Butler, "csp2b: A practical approach to combining CSP and B," *Formal Aspects of Computing*, vol. 12, no. 3, pp. 182–198, Nov 2000.

[10] D. Harel, "Statecharts: A visual formalism for complex systems," *Science of computer programming*, vol. 8, no. 3, pp. 231–274, 1987.

[11] D. Harel and A. Naamad, "The statemate semantics of statecharts," *ACM Trans. Softw. Eng. Methodol.*, vol. 5, no. 4, pp. 293–333, Oct. 1996.

[12] M. Frappier, F. Gervais, R. Laleau, B. Fraikin, and R. St-Denis, "Extending statecharts with process algebra operators," *Innovations in Systems and Software Engineering*, vol. 4, no. 3, pp. 285–292, Oct 2008.

[13] J. Milhau, M. Frappier, F. Gervais, and R. Laleau, "Systematic translation rules from ASTD to Event-B," in *Integrated Formal Methods*. Springer Berlin Heidelberg, 2010, pp. 245–259.

[14] M. Frappier, F. Gervais, R. Laleau, and J. Milhau, "Refinement patterns for ASTDs," *Formal Aspects of Computing*, vol. 26, no. 5, pp. 919–941, Sep 2014.

[15] T. Fayolle, "Combinaison des methodes formelles pour la specification de systemes industriels," Ph.D. dissertation, Department of Computer Science, Univ. of Paris Est Creteil, 2005.

[16] G. Hamon and J. Rushby, "An operational semantics for Stateflow," in *International Conference on Fundamental Approaches to Software Engineering*. Springer, 2004, pp. 229–243.

[17] M. Leuschel and M. J. Butler, "ProB: an automated analysis toolset for the B method," *STTT*, vol. 10, no. 2, pp. 185–203, 2008.

[18] M. Lu and J. Reeves, "Types of cyber attacks," Trustworthy Cyber Infrastructure for the Power Grid (TCIPG), Tech. Rep., 2018, https://tcipg.org/sites/default/files/rgroup/tcipg-reading-group-fall_2014_09-12.pdf.

[19] FireEye, "Anatomy of advanced persistent threats," https://www.fireeye.com/current-threats/anatomy-of-a-cyber-attack.html, 2018.

[20] L. N. Tidjon, M. Frappier, M. Leuschel, and A. Mammar, "Extended algebraic state-transition diagrams," University of Sherbrooke, Tech. Rep., 2018, https://depot.gril.usherbrooke.ca/fram1801/TR-25/blob/master/TR-25.pdf.

[21] M. Leuschel and E. Börger, "A compact encoding of sequential ASMs in Event-B," in *Proceedings ABZ 2016*, 2016, pp. 119–134.

[22] Formal Systems (Europe) Ltd, *Failures-Divergence Refinement — FDR User Manual (version 4.2)*. [Online]. Available: https://www.cs.ox.ac.uk/projects/fdr/manual/

[23] MITRE, "Att&ck - all techniques," https://attack.mitre.org/wiki/All_Techniques, 2018.

[24] R. A. Grimes, "Danger: Remote Access Trojans," https://technet.microsoft.com/en-us/library/dd632947.aspx, Apr. 2018, online, accessed 09/04/18.

[25] Microsoft, "System monitor (sysmon)," https://docs.microsoft.com/en-us/sysinternals/downloads/sysmon, 2017.

[26] CERT-EU, "Wannacry ransomware campaign exploiting smb vulnerability," CERT EU, Tech. Rep., 2017, https://cert.europa.eu/static/SecurityAdvisories/2017/CERT-EU-SA2017-012.pdf.

[27] Microsoft, "Windows registry information for advanced users," https://support.microsoft.com/en-ca/help/256986/windows-registry-information-for-advanced-users, 2012.

[28] M. Butler and M. Leuschel, "Combining CSP and B for specification and property verification," in *Proceedings of Formal Methods 2005*, ser. LNCS 3582. Springer-Verlag, 2005, pp. 221–236.

[29] M. Frappier, B. Fraikin, R. Chossart, R. Chane-Yack-Fa, and M. Ouenzar, "Comparison of model checking tools for information systems," in *Proceedings ICFEM*, 2010, pp. 581–596.

[30] K. Salabert, "iASTD, un interpréteur d'ASTD," *Master Thesis, Department of Computer Science, Univ. de Sherbrooke.*, 2011.

[31] B. Fraikin and M. Frappier, "Efficient symbolic execution of large quantifications in a process algebra," in *Formal Methods and Software Engineering*. Springer Berlin Heidelberg, 2007, pp. 327–344.