Technical Report 24 / Rapport Technique 24

version 1.3 2017-09-11

Département d'informatique

Faculté des sciences

UNIVERSITÉ DE SHERBROOKE

Algebraic State Transition Diagrams

Marc Frappier[1], Frédéric Gervais[2], Régine Laleau[2], Benoît Fraikin[1]

[1] GRIL, Département d'informatique, Université de Sherbrooke
Sherbrooke (Québec), J1K 2R1, Canada
{marc.frappier,benoit.fraikin}@usherbrooke.ca

[2] LACL, Université Paris-Est
IUT Fontainebleau, 77300 Fontainebleau, France
{frederic.gervais,laleau}@u-pec.fr

Abstract

This paper introduces a graphical notation called algebraic state transition diagrams *(ASTD), which allows for the combination of state transition diagrams using classical process algebra operators like sequence, iteration, parallel composition, quantified choice and quantified synchronization. It is inspired from automata, statecharts and process algebras. Hence, it combines the strength of all these notations: graphical representation, hierarchy, orthogonality, compositionality, abstraction. Quantification is one of the salient features of ASTDs, because it provides a powerful mechanism for modeling an arbitrary number of instances of an ASTD. A formal operational semantics is given. Our target application domain is the specification of information systems, but ASTDs are presented in a generic manner.*

Keywords. *State transition diagrams, statecharts, process algebras, information systems,* EB^3.

1 Introduction

Our aim is the formal specification of information systems (IS), and in particular, the specification of database applications. In previous work, we have studied the use of process algebras like EB^3 [13] to model dynamic properties of IS. The idea of using *state transition diagrams* to specify IS was also appealing to us, but we were unsatisfied with the capabilities of existing notations like statecharts [14, 15] and UML activity diagrams and state machine diagrams [20], because of the difficulty of explicitly and concisely representing multiple instances of an entity type and their interactions in an IS. For instance, it is easy to describe the behavior of a member borrowing a book in a library. However, it is very difficult to precisely describe how several members behave altogether to borrow and reserve books. To do so, one must use internal state variables and thus completely hide into event guards the ordering constraints, for instance between the creation of a member and the loans of the member, loosing the visual expressiveness of state transition diagrams. The connection between members and books over loans and reservations is even more difficult to model. In practice, specifiers will describe the single instance scenario (one member, one book) and let the implementer figure out the general case (several members and several books), given some natural language complementary description.

Interestingly, the interaction between several instances is easy to model using process algebras like CSP [16, 21] and EB^3: in [13], we have identified and specified the most typical patterns used in IS. Synchronization and quantification (also called indexing) allow for an elegant, formal, concise and complete representation of these scenarios. Hence, came the need of combining the visual expressiveness of state transition diagrams with the abstraction power of process algebras.

In this paper, we introduce a graphical notation called *algebraic state transition diagrams* (ASTD), which allows for the combination of state transition diagrams [9] using process algebra operators like sequence, iteration, parallel composition, quantified choice and quantified synchronization. It is inspired from automata [2], statecharts and process algebras. Hence, it leverages the strength of these notations: graphical representation, hierarchy, orthogonality, compositionality, abstraction. Quantification is one of the salient features of ASTDs, because it provides a powerful mechanism for modeling an arbitrary number of instances of an ASTD.

ASTDs support most of the main features of statecharts like hierarchy, OR-states, AND-states, guards and history states, but intentionally leaves out other features: i) no broadcast communication (ASTD use instead event synchronisation as in CSP [16, 21]), ii) no actions (ASTD only describe event traces), and iii) no null transition, i.e., transitions without event labels (an ASTD transition is always triggered by the reception of an event from the environment; each automaton transition is labeled by an event). We use ASTDs to describe the valid sequences of inputs that an IS must accept. They provide a convenient, precise (formal) and comprehensive way of representing all usage scenarios of an IS. IS outputs are not specified in an ASTD, because it is simpler and easier to specify them using attributes based on the traces of the ASTD, as in the EB^3 method [13].

As in ARGOS [18], automata constitute the ground term for ASTD construction. Automaton states can be elementary or a complex ASTD. However, ARGOS only includes parallel composition; ASTD includes all typical process algebra operators. In [5], a graphical notation inspired from Live Sequence Charts and Message Sequence Charts [8] is defined for dealing with event ordering on objects from a class, but it only supports quantified interleaving without synchronization. In [17], a process algebra semantics to Statecharts called SPL (Statecharts Process Language) is provided, without extending statecharts with process algebra operators. ASTDs also differ from algebraic state machines [7], which essentially represent states of a traditional state machine using an algebra [22]. In [10], single-user scenarios are represented as state-based relations depicted using state transition diagrams and integrated using a refinement-lattice meet operator.

ASTDs are closely related to process algebras like CSP [16], CCS [19], ACP [3], LOTOS [4] and EB^3 [13]. Essentially, ASTDs are like a process algebra with hierarchical automata as elementary process expressions. Automata can be combined freely with process algebra operators. ASTDs have a structured operational semantics in the Plotkin style, which has been first used by Milner for CCS and later on for LOTOS and CSP [21]. CSP also has a denotational semantics, given by traces, failures and divergences of a process. ACP is a true algebra, that is, its operators are first defined by a set of equations relating process algebra operators. CSP also includes a set of equations, on top of the denotational and operational semantics. LOTOS includes an algebraic notation for specifying abstract data types that are used in process expressions for data exchange. In ASTDs, we use attributes defined on the ASTD traces, as in the EB^3 notation. The attributes are defined using basic types which are assumed to be given

Model oriented notations, like B [1], Z [23] and ASM [6], are orthogonal to ASTDs and process algebras. The ordering of events is expressed by operation preconditions in the former, while it is expressed by a graph (automaton) and operators in the latter, which makes the ordering more explicit. Circus [24] combines the Z notation with CSP, which also makes event ordering more explicit; however, it does not include an automata-like notation.

The paper is organized as follows. Section 2 briefly describes basic types and typing conventions used in the paper. Then, we present the definition of ASTD types and states in Section 3. Section 4 shows an application of ASTDs to our (perennial!, we apologize) case study, a small library system. Finally, Section 5 concludes the paper with an appraisal of ASTDs and an outlook of future work.

2 Conventions and Definitions

The pointwise extension of a function $f \in S \nrightarrow T$ is the application of f to a set $S' \subseteq S$. It is noted $f[S']$ and defined as follows

$$f[S'] \triangleq \{f(s) \mid s \in S'\}$$

Let $T_1, \ldots, T_n$ be types. A type A defined as a *Cartesian product* is noted $A \triangleq \langle c_1, \ldots, c_n \rangle$ where $c_1 \in T_1, \ldots, c_n \in T_n$, which is usually represented as $A = T_1 \times \ldots \times T_n$ in the literature. Given an element $a \in A$, we write $a.c_i$ to denote the i^{th} coordinate of tuple a.

A *sum* is noted $B = \langle cons_1, A_1 \rangle \mid \ldots \mid \langle cons_m, A_m \rangle$, where each A_i is a (possibly empty) Cartesian product.

Symbol $cons_i$ denotes a sum tag (also called a constructor). Given an element $b \in B$, we write $b = (cons_i, c_{i_1}, \ldots, c_{i_n})$ to determine its sum subtype and to access its coordinates c_{i_j}. Parentheses "(" and ")" and brackets "[" and "]" may be used to determine precedence. By abuse of notation and for the sake of concision, we sometimes omit a coordinate name in a Cartesian product and directly use a sum type, when the sum type is more convenient. For example $(\ldots, [\perp \mid \top], \ldots)$ is a shorthand for $(\ldots, c_1, \ldots)$ where $c_1 \in \{\perp, \top\}$..

We use the following basic types. $\mathsf{Boolean}$ denotes the set $\{\mathsf{true}, \mathsf{false}\}$. Name denotes the set of state names. It includes two special elements, noted H and H^*, which respectively denote the shallow history state and the deep history state of statecharts [15]. Term denotes the set of terms constructed using types supported by the ASTD specification language. It is left undefined at this point, but it should include classical specification types like Boolean, integer, string, relations, functions, sequences, Cartesian product, sum, etc. Var denotes the set of variables. Event denotes the set of events that the system accepts. An event is noted $l(v_1, \ldots, v_n)$ where l is called the event label, and $v_i \in \mathsf{Term}$ are event parameters. Function α extracts the label of an event: $\alpha(l(v_1, \ldots, v_n)) = l$. Label denotes the set of event labels. $\mathsf{Predicate}$ denotes the set of first order predicates. Env denotes the set of environments. An environment is a function which maps a variable to a value; hence it is a set of pairs x_i, v_i, with $x_i \in \mathsf{Var}$ and $v_i \in \mathsf{Term}$. For convenience, an environment is noted $(\![x_1, \ldots, x_n := v_1 \ldots, v_n]\!)$, or, more concisely, $(\![\vec{x} := \vec{v}]\!)$. An empty environment is noted $(\![]\!)$.

An environment Γ can be used in a substitution. The expression $u[(\![\vec{x} := \vec{v}]\!)]$ denotes the simultaneous substitution of $x_1, \ldots, x_n$ by $v_1, \ldots, v_n$ in expression u, which can be a predicate or a term. The symbol $\lhd$ is a composition operator on environments such that $u[\Gamma_1 \lhd \Gamma_2] = (u[\Gamma_1])[\Gamma_2]$. Note that Γ_1 has precedence over Γ_2 when $\Gamma_1 \lhd \Gamma_2$ is used in a substitution.

3 ASTD

We denote by ASTD the type of all ASTDs. It includes the following subtypes: $\mathsf{Automaton}$, $\mathsf{Sequence}$, Guard, $\mathsf{Closure}$, Choice, $\mathsf{Synchronization}$, $\mathsf{QChoice}$, $\mathsf{QSynchronization}$, $\mathsf{ASTDCall}$, $\mathsf{ASTDElem}$. We shall describe each of them in the sequel. But first, we need to define some auxiliary notations.

ASTD subtypes share common concepts. Each $\mathsf{a} \in \mathsf{ASTD}$ has a set of states $\mathsf{a}.S \subseteq \mathsf{State}$. It is inductively defined. Some elements of S are said to be final: they enable subsequent work to start. Final states of an ASTD a are determined by a function *final* of type $\mathsf{State} \to \mathsf{Boolean}$. Function *init* of type $\mathsf{ASTD} \to \mathsf{State}$ returns the initial state of an ASTD. A state is either elementary or compound (another ASTD).

The semantics of ASTDs is defined in an operational style. It consists of a labeled transition system, which is a subset of $\mathsf{State} \times \mathsf{Event} \times \mathsf{State}$ and is inductively defined by inference rules. Elements of this relation are called *transitions* and noted $s \xrightarrow{\sigma}_{\mathsf{a}} s'$, which means that an ASTD a can execute event σ from state s and move to state s'. Subscript a can be omitted when it is clear from the context which ASTD is being referred to.

Because we use variables in some ASTD structures like quantified ASTDs and ASTD calls, we need the notion of an execution environment Γ, and we write transitions with respect to Γ, noted as $s \xrightarrow{\sigma,\Gamma}_{\mathsf{a}} s'$. We compute a transition starting from an empty environment, using the following inference rule.

$$\mathsf{env}\ \frac{s \xrightarrow{\sigma,(\![]\!)} s'}{s \xrightarrow{\sigma} s'}$$

ASTD are *nondeterministic*. If several transitions on σ are possible for a given state s, then one is nondeterministically chosen. The operational semantics is inductively defined in the sequel for each ASTD subtype.

3.1 Elementary ASTD

An elementary ASTD is used to define elementary states of an automaton ASTD. Formally, let $\mathsf{ASTDElem} \triangleq \langle \mathsf{elem} \rangle$ be the set of ASTD elem. The type of an ASTD elem state is $\langle \mathsf{elem}_\circ \rangle$. The initial and final states are as follows.

$$\begin{array}{rcl} \mathit{init}((\mathsf{elem})) & \triangleq & (\mathsf{elem}_\circ) \\ \mathit{final}((\mathsf{elem}_\circ)) & \triangleq & \mathsf{true} \end{array}$$

3.2 Automaton

An Example. An ASTD automaton is very similar to a traditional automaton, except that its states can be of any ASTD type, and that its transition function can refer to substates of automaton states, as in statecharts. Figure 1 provides a graphical representation of an example automaton named $\mathsf{a1}$. It includes a sub-automaton $\mathsf{4}$. The outer box delineates the definition of $\mathsf{a1}$. The tab of this box starts with the name of the automaton, with its parameter, x, of type int (integer). The name in the tab can be omitted. The keyword aut denotes that $\mathsf{a1}$ is an ASTD of type $\mathsf{Automaton}$. The initial state of an automaton is depicted by $\rightarrow\!\bigcirc$. For $\mathsf{a1}$, the initial state is $\mathsf{0}$, which is an elementary state (denoted by $\bigcirc$). An initial state could also be of any ASTD type; there are no restriction. Transitions are labeled in the statecharts style by $\mathsf{e}(\vec{x})[\phi]$, where $\mathsf{e}(\vec{x})$ is an event with parameters $\vec{x}$ and ϕ is a guard which must hold for the transition to trigger. Note that the statechart notion of *action* is not used

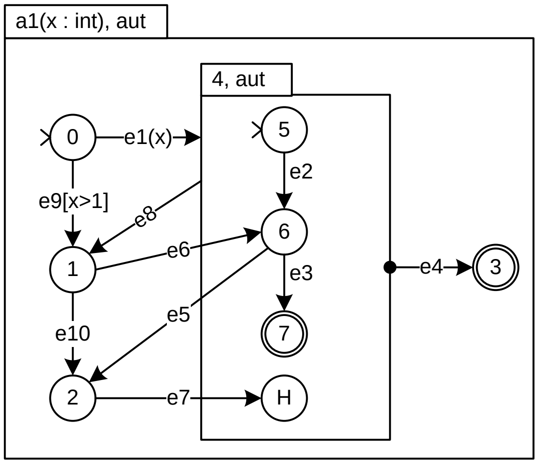

Figure 1. An automaton including another automaton

in this version of ASTD; however, it could be easily added, together with state variables. The event is mandatory on a transition and the guard is optional. A transition fires when an event is received from the environment and there exists a transition for that event in the current state of the automaton. If there is no transition in the current state for that event, it is ignored and discarded. In the context of IS, a meaningful error message should be provided to the environment (e.g., the user) when an event is discarded, Otherwise, the behavior of an automaton is essentially the same as the behavior of an OR-state in statecharts.

The states of an automaton are of type $\langle \mathsf{aut}_\circ, n, h, s \rangle$ where

- $n \in \mathsf{Name}$ denotes the name of the state.
- $h \in \mathsf{Name} \nrightarrow \mathsf{State}$ is a partial function that denotes the last visited substate of an automaton; it is used to implement the notion of *history* state introduced in statecharts.
- $s \in \mathsf{State}$ is the current state of the automaton. It can be a compound state, denoted by type State, or an elementary state, denoted by elem.

Suppose that a1 is instantiated with value $x := 2$. It is then in the initial state 0. The reception of the event $\mathsf{e1}(2)$ triggers a transition from 0 to state 4, which is a complex state given by an automaton. This puts 4 in its initial state 5. We denote this transition by

$$(\mathsf{aut}_\circ, 0, h, \mathsf{elem}_\circ) \xrightarrow{\mathsf{e1}(2)}_{\mathsf{a1}} (\mathsf{aut}_\circ, 4, h', (\mathsf{aut}_\circ, 5, h'', \mathsf{elem}_\circ))$$

For the sake of concision and illustration, let us simplify our notation for the moment and abstract from the type constructor $\mathsf{aut}_\circ$, history functions h, h', h'' and state type, by abbreviating this transition as $0 \xrightarrow{\mathsf{e1}(2)} 4(5)$. ASTD a1 can now accept event e2 and make the transition $4(5) \xrightarrow{\mathsf{e2}} 4(6)$, or accept e8 and make the transition $4(5) \xrightarrow{\mathsf{e2}} 1$. Suppose e2 has been accepted and then that e5 and e7 are accepted. If we summarize the transitions from the initial state, we have

$$0 \xrightarrow{\mathsf{e1}(2)} 4(5) \xrightarrow{\mathsf{e2}} 4(6) \xrightarrow{\mathsf{e5}} 2 \xrightarrow{\mathsf{e7}} 4(6)$$

The last transition (on e7) goes to the history state H. This means that it returns to the last visited state of 4, which is 6. Another path is

$$0 \xrightarrow{\mathsf{e9}} 1 \xrightarrow{\mathsf{e6}} 4(6) \xrightarrow{\mathsf{e3}} 4(7) \xrightarrow{\mathsf{e8}} 1 \xrightarrow{\mathsf{e10}} 2 \xrightarrow{\mathsf{e7}} 4(7)$$

since the history state points to 4(7) in that case. Hence, to manage the notion of history state, we must include in an automaton state a function h which stores the last visited substate of each state name. This function is stored in the state of a1 and is updated when 4 is left. Its initial value maps 4 to 5, the initial state of 4, :

$$h_{init} \triangleq \{4 \mapsto 5\}$$

Over transitions, h evolves as follows, noting state as (n, h), where n is the state name.

$$(0, \{4 \mapsto 5\}) \xrightarrow{\mathsf{e1}(2)} (4(5), \{4 \mapsto 5\}) \xrightarrow{\mathsf{e2}} (4(6), \{4 \mapsto 5\}) \xrightarrow{\mathsf{e8}} (1, \{4 \mapsto 6\})$$

Note that only transitions leaving 4 change the value of $h(4)$.

The transition labeled by e8 can be triggered whatever is the state of 4. The transition labeled by e4 is decorated by a bullet (•) at its source: this means that it can be fired only if 4 is in a final state. An elementary final state is denoted by ◎. Hence, the only possible transition to 3 is $4(7) \xrightarrow{\mathsf{e4}} 3$.

Formal Definition. Let $\mathsf{Automaton} \triangleq \langle \mathsf{aut}, name, \Sigma, N, \nu, \delta, SF, DF, n_0 \rangle$ be the set of automaton ASTDs. Note that we distinguish between a state of an ASTD and the ASTD itself. Each has its own type; by convention, we use subscript $_\circ$ (e.g., $\mathsf{aut}_\circ$) for the state type constructor. We have the following typing constraints on the components of an automaton.

- $name \in N$ is the name of the ASTD structure.
- $\Sigma \subseteq \mathsf{Event}$ is the alphabet.
- $N \subseteq \mathsf{Name} - \{\mathsf{H}, \mathsf{H}^*\}$ is the set of state names.
- $\nu \in N \rightarrow \mathsf{ASTD}$ maps each state name to ASTD, which can be elementary or complex.
- $\delta \subseteq \langle \eta, \sigma, \phi, \mathit{final}? \rangle$ is the transition relation, where:
  - η denotes the arrow. There are three types of arrows: $\langle \mathsf{loc}, n_1, n_2 \rangle$ denotes a transition from n_1 to n_2, $\langle \mathsf{tsub}, n_1, n_2, n_{2_\flat} \rangle$ denotes a transition from n_1 to substate $n_{2_\flat}$ of n_2 such that $\nu(n_2) \in \mathsf{Automaton}$, and $\langle \mathsf{fsub}, n_1, n_{1_\flat}, n_2 \rangle$ denote a transition from substate $n_{1_\flat}$ of n_1 to n_2 such that $\nu(n_1) \in \mathsf{Automaton}$.

- $\sigma \in$ Event.
- $\phi \in$ Predicate is the transition guard.
- $final? \in$ Boolean denotes a transition leaving from a final state (i.e., a transition annotated with a "•" at its source).

- $SF \subseteq N$ denotes the names of shallow final states.
- $DF \subseteq N$ denotes the names of deep final states, with $DF \cap SF = \emptyset$.
- $n_0 \in N$ is the name of the initial state.

Note that transitions to and from a substate are only allowed for automaton states, by conditions $\nu(n_2) \in$ Automaton and $\nu(n_1) \in$ Automaton. This differs from statecharts and UML statemachines, which allow transitions from and to substates of an AND-state. We made this choice to keep the syntax and the semantics simple. Guards can be used if such transitions are needed.

We now illustrate this formal definition by providing the textual representation of the example of Figure 1, whose declaration is $\mathsf{a1}(x : \mathsf{int}) \in$ Automaton The scope of x is automaton a1, which includes all its component ASTD which are locally declared in a1. The alphabet of a1 includes all the events that appear on transitions: $\mathsf{a1}.\Sigma \triangleq \{\mathsf{e1}, \mathsf{e4}, \mathsf{e5}, \mathsf{e6}, \mathsf{e7}, \mathsf{e8}, \mathsf{e9}, \mathsf{e10}\}$. Note that e2 and e3 are internal to automaton 4; hence they belong to the alphabet of 4. The state names of a1 are $\mathsf{a1}.N \triangleq \{0, 1, 2, 3, 4\}$ and they are mapped as follows

$$\begin{aligned}\mathsf{a1}.\nu \triangleq\ & \{0 \mapsto (0, \mathsf{elem}_\circ), 1 \mapsto (1, \mathsf{elem}_\circ), 2 \mapsto (2, \mathsf{elem}_\circ),\\ & 3 \mapsto (3, \mathsf{elem}_\circ), 4 \mapsto 4\}\end{aligned}$$

Names 0,1,2,3 are mapped to elementary states; name 4 is mapped to the sub-automaton 4. The transition relation $\mathsf{a1}.\delta$ contains the following transitions.

$$\begin{array}{l}
\delta(\ (\mathsf{loc}, 0, 4) \quad, \mathsf{e1}(x), \mathsf{true} \quad, \mathsf{false}\ )\\
\delta(\ (\mathsf{loc}, 4, 3) \quad, \mathsf{e4} \quad, \mathsf{true} \quad, \mathsf{true}\ )\\
\delta(\ (\mathsf{fsub}, 4, 6, 2), \mathsf{e5} \quad, \mathsf{true} \quad, \mathsf{false}\ )\\
\delta(\ (\mathsf{tsub}, 1, 4, 6), \mathsf{e6} \quad, \mathsf{true} \quad, \mathsf{false}\ )\\
\delta(\ (\mathsf{tsub}, 2, 4, \mathsf{H}), \mathsf{e7} \quad, \mathsf{true} \quad, \mathsf{false}\ )\\
\delta(\ (\mathsf{loc}, 4, 1) \quad, \mathsf{e8} \quad, \mathsf{true} \quad, \mathsf{false}\ )\\
\delta(\ (\mathsf{loc}, 0, 1) \quad, \mathsf{e9} \quad, x > 1, \mathsf{false}\ )\\
\delta(\ (\mathsf{loc}, 1, 2) \quad, \mathsf{e10} \quad, \mathsf{true} \quad, \mathsf{false}\ )
\end{array}$$

The shallow final states of a1 are $\mathsf{a1}.SF \triangleq \{3\}$. There are no deep final states in a1. Elementary final states (denoted by ◎) are always considered as shallow, since they do not contain a sub-ASTD. Note that state 4 is not a final state; its automaton component 4 includes a final state, but that does not make 4 a final state. The initial state of a1 is $\mathsf{a1}.n_0 \triangleq 0$. Automaton 4 is described in a similar manner.

Operational Semantics. Functions *final* and *init* determine, respectively, if a state is final and the initial state of an ASTD.

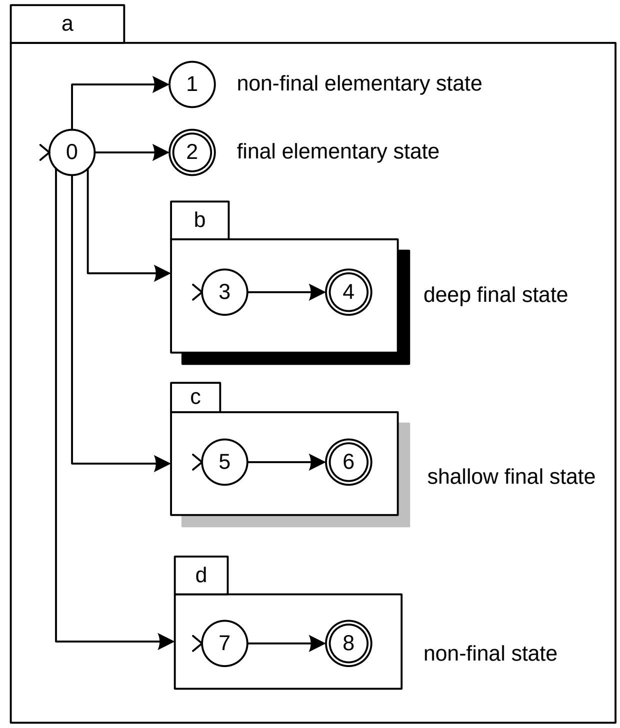

Figure 2. Graphical representation of final states

$$\begin{array}{rcl}
init((\mathsf{aut}, \ldots)) & \triangleq & (\mathsf{aut}_\circ, n_0, h_{init}, init(\nu(n_0)))\\
h_{init} & \triangleq & \{n \mapsto init(\nu(n)) \mid n \in N\}\\
final((\mathsf{aut}_\circ, n, h, s)) & \triangleq & (n \in DF \wedge final(s))\\
 & & \vee\\
 & & (n \in SF)
\end{array}$$

Note that we must use the full description of a state, for the sake of completeness. The initial state of an automaton is the state named n_0. Its history function is initialized by mapping each state name to the initial state of its internal structure: elementary states are mapped to the constant elem; ASTD state names are mapped to the initial state of their ASTD, recursively. An ASTD state is final if it is one of the shallow final states or if it is a deep final state and its internal state is final (recursively). Complex shallow final states are denoted by a grey shaded box; deep final states are denoted by a black shaded box, as illustrated in Figure 2.

There are six rules of inference, written in the usual form $\frac{premiss}{conclusion}$. The first rule, aut_1, describe a transition between local states.

$$\mathsf{aut}_1 \ \frac{\delta((\mathsf{loc}, n_1, n_2), \sigma', g, final?) \qquad \Psi}{(\mathsf{aut}_\circ, n_1, h, s) \xrightarrow{\sigma, \Gamma} (\mathsf{aut}_\circ, n_2, h', init(\nu(n_2)))}$$

Recall that the ASTD semantics is a transition relation on State. The transition relation δ of an automaton is simply defined on state names from N. Inference rule aut_1 describes how δ relates to the overall state transition relation, taking into account the history function and the arbitrary type of automaton states (elementary or ASTD). The target state of the transition is the initial state of the destination state in δ: for an elementary state, recall that $init(\langle \mathsf{elem} \rangle) = \mathsf{elem}_\circ$; for an ASTD state, $init$ returns the particular initial state of that structure. This shall become more obvious when other ASTD types are described in the sequel. Five rules share a common premiss, which we abbreviate by Ψ.

$$\Psi \triangleq \big((\mathit{final}? \Rightarrow \mathit{final}(s)) \wedge g \wedge \sigma' = \sigma \wedge h' = h \mathbin{\lhd\!\!-} \{n_1 \mapsto s\}\big)[\Gamma]$$

It provides that a transition noted as *final*? must start from a final state, that the transition guard g holds, and that the event received, noted σ, is equal, under the current transition environment Γ, to the event specified in the transition relation, noted σ'. Moreover, the history function in the target state, noted h', is updated by storing the last visited substate of n_1. It is defined using operator $\mathbin{\lhd\!\!-}$, the override operator of the B and Z notation.

Rule aut_2, handles transitions to substates, in the particular case where the substate is not an history state.

$$\mathsf{aut}_2 \; \frac{\begin{array}{c}\delta((\mathsf{tsub}, n_1, n_2, n_{2_\flat}), \sigma', g, \mathit{final}?) \\ n_{2_\flat} \notin \{\mathsf{H}, \mathsf{H}^*\} \\ \Psi\end{array}}{(\mathsf{aut}_\circ, n_1, h, s) \xrightarrow{\sigma,\Gamma} (\mathsf{aut}_\circ, n_2, h', (\mathsf{aut}_\circ, n_{2_\flat}, h_{init}, init(\nu(n_{2_\flat})))))}$$

The target state is n_2, with $n_{2_\flat}$ as its substate. Again, the initial state of the substate is targeted (since this substate could also be an ASTD).

Rule aut_3 handles transitions to a *shallow* history state (noted H), following the behavior prescribed by statecharts.

$$\mathsf{aut}_3 \; \frac{\begin{array}{c}\delta((\mathsf{tsub}, n_1, n_2, \mathsf{H}), \sigma', g, \mathit{final}?) \\ n_{2_\flat} = name(h(n_2)) \\ \Psi\end{array}}{(\mathsf{aut}_\circ, n_1, h, s) \xrightarrow{\sigma,\Gamma} (\mathsf{aut}_\circ, n_2, h', (\mathsf{aut}_\circ, n_{2_\flat}, h_{init}, init(\nu(n_{2_\flat})))))}$$

Function $name$ returns the name of an automaton state: $name(\mathsf{aut}_\circ, n, \ldots) = n$. In the case of shallow history, the target state is the *initial state* of the ASTD referenced by $h(n_1)$.

Rule aut_4 handles transitions to a *deep* history state (noted H*); in that case, the target state is the full state recorded in $h(n_2)$.

$$\mathsf{aut}_4 \; \frac{\delta((\mathsf{tsub}, n_1, n_2, \mathsf{H}^*), \sigma', g, \mathit{final}?) \qquad \Psi}{(\mathsf{aut}_\circ, n_1, h, s) \xrightarrow{\sigma,\Gamma} (\mathsf{aut}_\circ, n_2, h', h(n_2))}$$

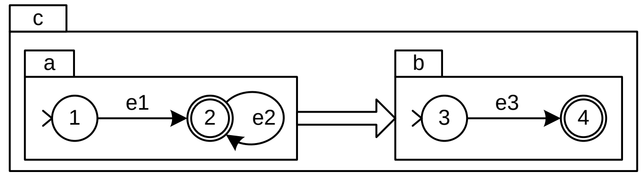

Figure 3. A sequence ASTD including two simple automata

Rule aut_5 handles transitions from a substate.

$$\mathsf{aut}_5 \; \frac{\begin{array}{c}\delta((\mathsf{fsub}, n_1, n_{1_\flat}, n_2), \sigma', g, \mathit{final}?) \\ name(s) = n_{1_\flat} \\ \Psi\end{array}}{(\mathsf{aut}_\circ, n_1, h, s) \xrightarrow{\sigma,\Gamma} (\mathsf{aut}_\circ, n_2, h', init(\nu(n_2)))}$$

Rule aut_6, handles transitions within a substate.

$$\mathsf{aut}_6 \; \frac{s \xrightarrow{\sigma,\Gamma}_{\nu(n)} s'}{(\mathsf{aut}_\circ, n, h, s) \xrightarrow{\sigma,\Gamma} (\mathsf{aut}_\circ, n, h, s')}$$

This is the first recursive rule where the compositionality of our semantics is illustrated. It requires to prove that σ can be executed in the substate, which could be any ASTD. In the target state of the conclusion, only the substate of the automaton state is changing; the automaton says in the same state name. The history function is unchanged.

3.3 Sequence

The sequence ASTD is a new concept with respect to statecharts. It allows for the sequential composition of two ASTDs. When the first one reaches a final state, the second one can start its execution. This is particularly useful for problems which can be decomposed into a set of tasks that have to be executed in sequence.

An Example. Figure 3 illustrates a very simple sequence ASTD, whose component ASTDs are two simple automata. Automaton a, which is on the left-hand side (LHS) of the arrow symbol, is the first to execute. Upon reception of event e1, it makes a transition from 1 to 2 and reaches a final state. This enables event e3 in b to be executed upon its reception. Event e2 is also executable, since it appears on a transition from 2. Suppose e3 is received. Then the sequence ASTD c leaves ASTD a and executes e3 on b. To represent these transitions, we first need to defined the type of a sequence state, which is $\langle \Rightarrow_\circ, [\mathsf{fst} \mid \mathsf{snd}], s\rangle$, where $s \in \mathsf{State}$. Keyword left indicates that the sequence ASTD is in its LHS state, and dually for right. The sequence of

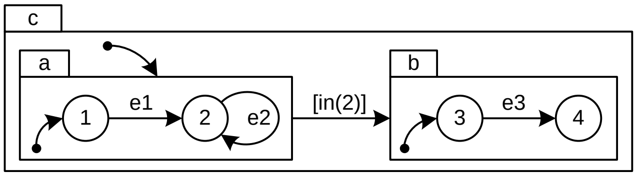

Figure 4. A statechart reproducing the sequence ASTD of Figure 3

events just described is represented as follows.

$$\begin{array}{ll} & (\Rightarrow_\circ, \mathsf{fst}, (\mathsf{aut}_\circ, 1, h, \mathsf{elem}_\circ)) \\ \xrightarrow{\mathsf{e1}}_{\mathsf{c}} & (\Rightarrow_\circ, \mathsf{fst}, (\mathsf{aut}_\circ, 2, h', \mathsf{elem}_\circ)) \\ \xrightarrow{\mathsf{e3}}_{\mathsf{c}} & (\Rightarrow_\circ, \mathsf{snd}, (\mathsf{aut}_\circ, 4, h'', \mathsf{elem}_\circ)) \end{array}$$

The notion of final state does no exist in statecharts. To reproduce in statecharts the same behavior as a sequence ASTD, one could use a guarded null transition between the two statecharts (see Figure 4); its guard is expressed using a predicate like $in(s_1) \vee \ldots in(s_n)$, where s_i is a state considered as final in the first statechart, thereby exhibiting the structure of the inner statecharts into the outer statechart, and increasing coupling between the two. If the inner statechart is more complex, things get even more complicated. Note also that the initial state of a sequence ASTD is simply the initial state of its first component. Hence, sequence is a useful abstraction fostering simplicity in specification design.

Formal Definition and Semantics. Let Sequence $\triangleq \langle \Rightarrow, n, \mathit{fst}, \mathit{snd} \rangle$ be the set of sequence ASTDs, where $\mathit{fst}, \mathit{snd} \in$ ASTD are respectively the first and second element of the sequence. Functions *init* and *final* are defined as follows.

$$\begin{array}{rcl} \mathit{init}((\Rightarrow, n, \mathit{fst}, \mathit{snd})) & \triangleq & (\Rightarrow_\circ, \mathsf{fst}, \mathit{init}(\mathit{fst})) \\ \mathit{final}((\Rightarrow_\circ, \mathsf{fst}, s)) & \triangleq & \mathit{final}(s) \wedge \mathit{final}(\mathit{init}(\mathit{snd})) \\ \mathit{final}((\Rightarrow_\circ, \mathsf{snd}, s)) & \triangleq & \mathit{final}(s) \end{array}$$

The initial state of a sequence ASTD is the initial state of its LHS ASTD. A sequence ASTD is in a final state if either of the following two cases holds: i) it is executing its LHS ASTD and this ASTD is in a final state, and the initial state of the RHS ASTD is also a final state; ii) it is executing the RHS ASTD which is in a final state.

We need three rules to define a sequence. Rule $\Rightarrow_1$ deals with transitions on the LHS ASTD only. Rule $\Rightarrow_2$ deals with transitions from the LHS to RHS, when the LHS is in a final state. Rule $\Rightarrow_3$ deals with transitions on the RHS ASTD.

$$\Rightarrow_1 \ \frac{s \xrightarrow{\sigma,\Gamma}_{\mathit{fst}} s'}{(\Rightarrow_\circ, \mathsf{fst}, s) \xrightarrow{\sigma,\Gamma} (\Rightarrow_\circ, \mathsf{fst}, s')}$$

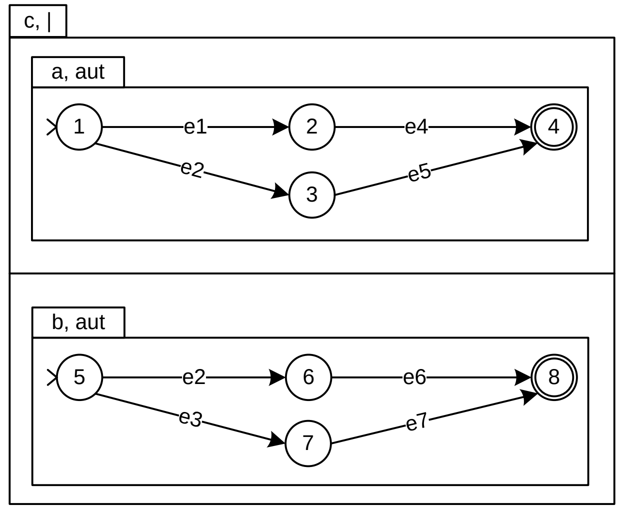

Figure 5. A choice ASTD including two automata

$$\Rightarrow_2 \ \frac{\mathit{final}(s)[\Gamma] \qquad \mathit{init}(\mathit{snd}) \xrightarrow{\sigma,\Gamma}_{\mathit{snd}} s'}{(\Rightarrow_\circ, \mathsf{fst}, s) \xrightarrow{\sigma,\Gamma} (\Rightarrow_\circ, \mathsf{snd}, s')}$$

$$\Rightarrow_3 \ \frac{s \xrightarrow{\sigma,\Gamma}_{\mathit{snd}} s'}{(\Rightarrow_\circ, \mathsf{snd}, s) \xrightarrow{\sigma,\Gamma} (\Rightarrow_\circ, \mathsf{snd}, s')}$$

3.4 Choice

A choice ASTD allows a choice between two component ASTDs. Once a component has been chosen, the other component is ignored. It is essentially the same as a choice operator in a process algebra. The choice is nondeterministic if each component can execute the requested event.

An example. Figure 5 provides an example of a choice ASTD, which includes two automata components. If e1 is received, then a is chosen to execute it. The subsequent events will be accepted by a only. Dually, if e3 is received, then b is chosen to execute it. If e2 is received, then a nondeterministic choice is made between a and b to execute it.

Formal Definition and Semantics. Let Choice $\triangleq \langle |, n, l, r \rangle$ be the set of choice ASTDs, where $l, r \in$ ASTD are respectively the first and second element of the choice. The type of a choice state is $\langle |_\circ, \mathit{side}, s \rangle$ where $\mathit{side} \in (\bot \mid \langle \mathsf{left} \rangle \mid \langle \mathsf{right} \rangle)$ denotes the component which has been chosen, and $s \in (\mathsf{State} \mid \bot)$ denotes the state of the component ASTD which has been chosen. In the initial state, it is defined as $\bot$. A choice state is final if i) it hasn't started yet and the initial state of each component is final, or ii) the chosen component is in a final state. Here are the formal definitions of the initial state and the final states.

$$\begin{array}{rcl} init((|, n, l, r)) & \triangleq & (|_\circ, \bot, \bot) \\ \mathit{final}((|_\circ, \bot, \bot)) & \triangleq & \mathit{final}(init(l)) \vee \mathit{final}(init(r)) \\ \mathit{final}((|_\circ, \mathsf{left}, s)) & \triangleq & \mathit{final}(s) \\ \mathit{final}((|_\circ, \mathsf{right}, s)) & \triangleq & \mathit{final}(s) \end{array}$$

There are four rules of inference. The first two deal with the execution of the first event from the initial state. The other two deal with execution of the subsequent events from the chosen component.

$$|_1 \; \frac{init(l) \xrightarrow{\sigma,\Gamma}_l s'}{(|_\circ, \bot, \bot) \xrightarrow{\sigma,\Gamma} (|_\circ, \mathsf{left}, s')}$$

$$|_2 \; \frac{init(r) \xrightarrow{\sigma,\Gamma}_r s'}{(|_\circ, \bot, \bot) \xrightarrow{\sigma,\Gamma} (|_\circ, \mathsf{right}, s')}$$

$$|_3 \; \frac{s \xrightarrow{\sigma,\Gamma}_l s'}{(|_\circ, \mathsf{left}, s) \xrightarrow{\sigma,\Gamma} (|_\circ, \mathsf{left}, s')}$$

$$|_4 \; \frac{s \xrightarrow{\sigma,\Gamma}_r s'}{(|_\circ, \mathsf{right}, s) \xrightarrow{\sigma,\Gamma} (|_\circ, \mathsf{right}, s')}$$

3.5 Kleene closure

This operator comes from regular expressions. It allows for iteration on an ASTD an arbitrary number of times (including zero). An iteration is completed when the component ASTD has reached a final state. At the end of an iteration, a Kleene closure can start a new iteration or be itself in a final state (and allow, for instance, an outer sequence ASTD to start the next task). This behavior is very common in IS. For instance, a typical pattern is the producer-modifier-consumer of an entity or an association. The user can iterate an arbitrary number of times on the modifiers and then terminate with a consumer. We shall illustrate that pattern in our small case study.

An Example. Figure 6 illustrates a closure applied to the a sequence ASTD similar to c of Figure 3, except that the LHS and RHS are also themselves within a closure. As a convention, we coalesce ASTD boxes when the outer ASTD is a unary operator, like Kleene closure; the coalescing is indicated by adding the tab of the inner ASTD to the outer unary ASTD (see Figure 8). The initial state of a closure is the initial state of its component ASTD. From its initial state, the closure f can execute either: e1 on a, or e3 on b, since the LHS of e is the closure c, which can terminate immediately and allow the RHS of e to execute e3 (i.e., the initial state of a closure is also a final state, to allow for 0 iteration). The statechart equivalent of this closure is shown

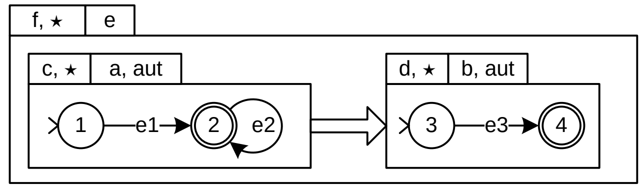

Figure 6. A closure ASTD including a sequence ASTD

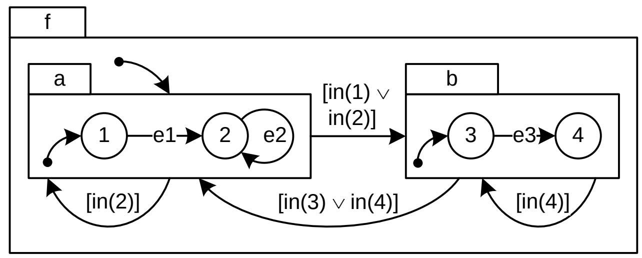

Figure 7. A statechart reproducing the ASTD of Figure 6

in Figure 7. It preserves the automaton decomposition into a and b, and adds null transitions in a systematic way to simulate the closure. Indeed, to simulate a closure, one must add a null transition from the final states to the initial state. In b, both states 3 and 4 are final, since there is a closure in Figure 6 on b. The guard of the transition between a and b, which simulates the sequential composition, must refer to both the initial and final states of the LHS of the sequence, since a closure allows for 0 iteration on a. This simple example illustrates that algebraic operators nicely encapsulate complex behavior compositions, compared to statecharts. Moreover, this example allows for an infinite sequence of null transitions (i.e., a divergence), which is annoying for a statechart interpreter, because it must detect these cases. This does not occur in an ASTD, because the operational semantics embodies the notion of final and initial states without inducing an infinite recursion.

Formal Definition and Semantics. Let $\mathsf{Closure} \triangleq \langle \star, n, b \rangle$ be the set of Kleene closure ASTDs, where $b \in \mathsf{ASTD}$ is the body of the closure. The type of a closure state is $\langle \star_\circ, started?, s \rangle$ where $s \in \mathsf{State}$ and $started? \in \mathsf{Boolean}$ indicates whether the first iteration has been started. It is essentially used to determine if the closure can immediately exit without any iteration. Initial and final states are defined

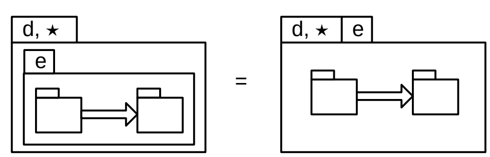

Figure 8. Unary ASTD box coalescing

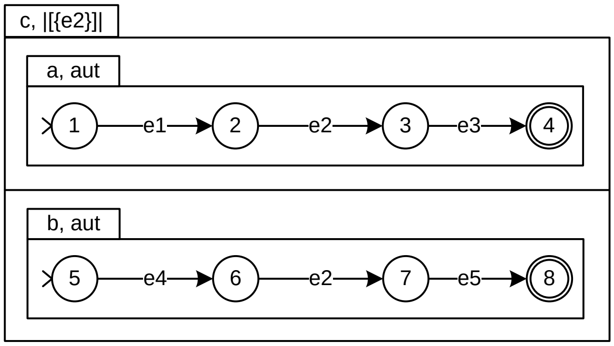

Figure 9. A synchronization ASTD including two automata

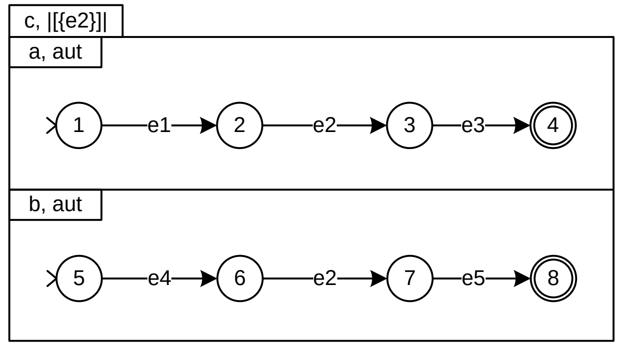

Figure 10. A coalesced version of the ASTD of Figure 9

as follows.

$$\begin{array}{rcl} \mathit{init}((\star, n, b)) & \triangleq & (\star_\circ, \mathsf{false}, \bot) \\ \mathit{final}((\star_\circ, \mathit{started?}, s)) & \triangleq & \mathit{final}(s) \vee \neg \mathit{started?} \end{array}$$

There are two inference rules: $\star_1$ allows for (re-)starting from the initial state of the component ASTD when a final state has been reached or for the first iteration; $\star_2$ allows for execution on the component ASTD when an iteration has already started.

$$\star_1 \ \frac{(\mathit{final}(s)[\Gamma] \vee \neg \mathit{started?}) \qquad \mathit{init}(b) \xrightarrow{\sigma,\Gamma}_b s'}{(\star_\circ, \mathit{started?}, s) \xrightarrow{\sigma,\Gamma} (\star_\circ, \mathsf{true}, s')}$$

$$\star_2 \ \frac{s \xrightarrow{\sigma,\Gamma}_b s'}{(\star_\circ, \mathsf{true}, s) \xrightarrow{\sigma,\Gamma} (\star_\circ, \mathsf{true}, s')}$$

3.6 Parameterized synchronization

The parameterized synchronisation is similar to an AND-state in statecharts, in that it allows two ASTDs to execute in parallel, but these two ASTD must synchronize on events whose label are in the synchronization set Δ. By synchronization, we mean that the two ASTDs must execute the event at the same time; there is no communication by message broadcasting. Events whose labels are not in Δ are executed in interleave. Thus, it is essentially the same behavior as the parameterized synchronization found in process algebra like Lotos or Roscoe's version of CSP [21]. As such, it also conveniently represents a conjunction of ordering constraints on events of Δ. When Δ is empty, it behaves like an interleave operation.

An Example. Figure 9 provides an example of a synchronization ASTD named c, with $\Delta = \{\mathsf{e2}\}$, which is noted $|[\{\mathsf{e2}\}]|$ in the tab. It includes two automata a and b. The initial state of c is the initial state of its components. From the initial state, c can execute either e1 or e4. After executing these two events (in any order), the two ASTDs a and b must execute e2 at the same time. Then, e3 and e5 can be executed in any order. The type of a synchronization state is $\langle |[]|_\circ, s_l, s_r \rangle$ where $s_l, s_r \in \mathsf{State}$. Here is a possible sequence of transitions, where _ denotes the history function which is omitted, for the sake of concision.

$$\begin{array}{ll} & (|[]|_\circ, (\mathsf{aut}_\circ, 1, \_, \mathsf{elem}_\circ), (\mathsf{aut}_\circ, 5, \_, \mathsf{elem}_\circ)) \\ \xrightarrow{\mathsf{e1}}_{\mathsf{c}} & (|[]|_\circ, (\mathsf{aut}_\circ, 2, \_, \mathsf{elem}_\circ), (\mathsf{aut}_\circ, 5, \_, \mathsf{elem}_\circ)) \\ \xrightarrow{\mathsf{e4}}_{\mathsf{c}} & (|[]|_\circ, (\mathsf{aut}_\circ, 2, \_, \mathsf{elem}_\circ), (\mathsf{aut}_\circ, 6, \_, \mathsf{elem}_\circ)) \\ \xrightarrow{\mathsf{e2}}_{\mathsf{c}} & (|[]|_\circ, (\mathsf{aut}_\circ, 3, \_, \mathsf{elem}_\circ), (\mathsf{aut}_\circ, 7, \_, \mathsf{elem}_\circ)) \\ \xrightarrow{\mathsf{e3}}_{\mathsf{c}} & (|[]|_\circ, (\mathsf{aut}_\circ, 4, \_, \mathsf{elem}_\circ), (\mathsf{aut}_\circ, 7, \_, \mathsf{elem}_\circ)) \\ \xrightarrow{\mathsf{e5}}_{\mathsf{c}} & (|[]|_\circ, (\mathsf{aut}_\circ, 4, \_, \mathsf{elem}_\circ), (\mathsf{aut}_\circ, 8, \_, \mathsf{elem}_\circ)) \end{array}$$

When an ASTD based on a binary operator like $|[\Delta]|$ includes an automaton component or a unary operator ASTD, we can also coalesce the component ASTD with its enclosing box from the binary operator. Figure 10 illustrates a coalesced version of the ASTD of Figure 9.

Formal Definition and Semantics. Let $\mathsf{Synchronization} \triangleq \langle |[]|, n, \Delta, l, r \rangle$ be the set of parameterized synchronization ASTDs, where $\Delta \subseteq \mathsf{Label}$ denotes a synchronization set of event labels and $l, r \in \mathsf{ASTD}$ are the synchronized ASTDs. Initial and final states are defined as follows.

$$\begin{array}{rcl} \mathit{init}((|[]|, n, \Delta, l, r))) & \triangleq & (|[]|_\circ, \mathit{init}(l), \mathit{init}(r)) \\ \mathit{final}((|[]|_\circ, s_l, s_r)) & \triangleq & \mathit{final}(s_l) \wedge \mathit{final}(s_r) \end{array}$$

There are three inference rules. Rules $|[]|_1$ and $|[]|_2$ respectively describe execution of events with no synchronization required on the LHS and the RHS of the synchronization ASTD. Rule $|[]|_3$ describe the synchronization between the LHS and the RHS.

$$|[]|_1 \ \frac{\alpha(\sigma) \notin \Delta \qquad s_l \xrightarrow{\sigma,\Gamma}_l s_l'}{(|[]|_\circ, s_l, s_r) \xrightarrow{\sigma,\Gamma} (|[]|_\circ, s_l', s_r)}$$

$$|[]|_2 \ \frac{\alpha(\sigma) \notin \Delta \qquad s_r \xrightarrow{\sigma,\Gamma}_r s_r'}{(|[]|_\circ, s_l, s_r) \xrightarrow{\sigma,\Gamma} (|[]|_\circ, s_l, s_r')}$$

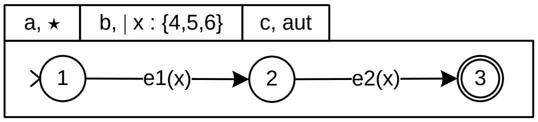

Figure 11. A closure over a quantified choice ASTD

$$|[]|_3 \; \frac{\alpha(\sigma) \in \Delta \qquad s_l \xrightarrow{\sigma,\Gamma}_l s_l' \qquad s_r \xrightarrow{\sigma,\Gamma}_r s_r'}{(|[]|_\circ, s_l, s_r) \xrightarrow{\sigma,\Gamma} (|[]|_\circ, s_l', s_r')}$$

We use the abbreviation $\| \triangleq |[]|$ with $\Delta = \alpha(l) \cap \alpha(r)$, where $\alpha(a)$ denotes the labels of event appearing in ASTD a, including all its inner ASTDs. It is the parallel composition operator of CSP, which means that the ASTDs synchronize on common events. We also use $||| \triangleq |[\{\}]|$ with $\Delta = \{\}$, which is the interleave operator of CSP.

3.7 Quantified choice

This operator and the next one (quantified synchronization) are not usual operators in state diagrams. They have been introduced to take into account IS specificities, like managing sets of entity type instances. The quantified choice is very similar to an existential quantification in first-order logic. It allows to pick a value from a set and execute a component ASTD with that value. The scope of the quantified variable is the component ASTD. Figure 11 illustrates a closure over a choice quantification of an automaton. ASTD a iterates on the choice. At each iteration, a new value for x is chosen. The choice quantification is represented by $\mid x : \{4, 5, 6\}$.

The type of a quantification choice state is $\langle |{:}_\circ, [\bot \mid v], [\bot \mid s] \rangle$ where $\bot$ is a constant indicating that the choice hasn't been made yet, and $v \in \mathsf{Term}$ denotes the current value of the choice quantified variable when the choice has been made.

The following is a possible sequence of transitions for the ASTD of Figure 11.

$$\begin{array}{ll} & (\star_\circ, \mathsf{false}, (|{:}_\circ, \bot, (\mathsf{aut}_\circ, 1, \_, \mathsf{elem}_\circ))) \\ \xrightarrow{\mathsf{e1}(5)}_{\mathsf{c}} & (\star_\circ, \mathsf{true}, (|{:}_\circ, 5, (\mathsf{aut}_\circ, 2, \_, \mathsf{elem}_\circ))) \\ \xrightarrow{\mathsf{e2}(5)}_{\mathsf{c}} & (\star_\circ, \mathsf{true}, (|{:}_\circ, 5, (\mathsf{aut}_\circ, 3, \_, \mathsf{elem}_\circ))) \\ \xrightarrow{\mathsf{e1}(4)}_{\mathsf{c}} & (\star_\circ, \mathsf{true}, (|{:}_\circ, 4, (\mathsf{aut}_\circ, 2, \_, \mathsf{elem}_\circ))) \end{array} \qquad \text{(TR1)}$$

In the initial state, special value $\bot$ is used to indicate that the quantified variable hasn't been instantiated yet. The quantified choice ASTD can accept $\mathsf{e1}(4)$, $\mathsf{e1}(5)$ and $\mathsf{e1}(6)$. When event $\mathsf{e1}(5)$ is received, the only value of x for which the quantified choice can accept $\mathsf{e1}(5)$ is $x = 5$. This value is recorded in the $|{:}_\circ$ state. The iteration can complete only by accepting event $\mathsf{e2}(5)$. In the next iteration, a new value of x can be chosen. Again, $\mathsf{e1}(4)$, $\mathsf{e1}(5)$ and $\mathsf{e1}(6)$ can be accepted. When $\mathsf{e1}(4)$ is received, x is bound to 4 and a new iteration can start.

Here is the semantics. Let $\mathsf{QChoice} \triangleq \langle |{:}, n, x, T, b \rangle$ be the set of quantified choice ASTDs, where $x \in \mathsf{Var}$ denotes a quantification variable, T is a type and $b \in \mathsf{ASTD}$ is the quantified ASTD. Initial and final states are defined as follows.

$$init((|{:}, n, x, T, b)) \triangleq (|{:}_\circ, \bot, \bot)$$

$$\mathit{final}((|{:}_\circ, \bot, \bot)) \triangleq \exists\, x : T \cdot \mathit{final}(init(b))$$

$$v \neq \bot \Rightarrow (\mathit{final}((|{:}_\circ, v, s)) \triangleq \mathit{final}(s)[x := v])$$

This is the first type of ASTD where we need the notion of environment, to manage the quantification. When a transition is computed using the inference rules, the value bound to the quantification variable is added to the execution environment (the one appearing on the transition arrow) and can be used to make the proof, in particular to check that the event received σ matches the transition event σ', after the environment has been applied as a substitution. This behavior is expressed by the following two inference rules.

$$|{:}_1 \; \frac{init(b) \xrightarrow{\sigma, \llbracket x := v \rrbracket \lhd \Gamma}_b s' \qquad v \in T}{(|{:}_\circ, \bot, \bot) \xrightarrow{\sigma,\Gamma} (|{:}_\circ, v, s')}$$

$$|{:}_2 \; \frac{s \xrightarrow{\sigma, \llbracket x := v \rrbracket \lhd \Gamma}_b s' \qquad v \neq \bot}{(|{:}_\circ, v, s) \xrightarrow{\sigma,\Gamma} (|{:}_\circ, v, s')}$$

We can illustrate them by proving the last transition of (TR1) (see previous page); we abbreviate true by T.

$$\mathsf{env}\;\dfrac{\star_1\;\dfrac{|{:}_1\;\dfrac{\mathsf{aut}_1\;\dfrac{\begin{array}{c}\mathsf{e1}(4) = \mathsf{e1}(x)[\llbracket x := 4 \rrbracket] \\ \delta((\mathsf{loc}, 1, 2), \mathsf{e1}(x), \mathsf{true}, \mathsf{false})\end{array}}{(1) \xrightarrow{\mathsf{e1}(4),\, \llbracket x := 4 \rrbracket}_{\mathsf{c}} (2)}}{(|{:}_\circ, \bot, (1)) \xrightarrow{\mathsf{e1}(4),\, \llbracket\rrbracket}_{\mathsf{b}} (|{:}_\circ, 4, (2))}}{(\star_\circ, \mathsf{T}, (|{:}_\circ, 5, (3))) \xrightarrow{\mathsf{e1}(4), \llbracket\rrbracket}_{\mathsf{a}} (\star_\circ, \mathsf{T}, (|{:}_\circ, 4, (2)))}}{(\star_\circ, \mathsf{T}, (|{:}_\circ, 5, (3))) \xrightarrow{\mathsf{e1}(4)} (\star_\circ, \mathsf{T}, (|{:}_\circ, 4, (2)))}$$

A lemma implicitly used in this proof in step $\star_1$ is that b is in a final state.

$$\dfrac{\dfrac{3 \in \mathsf{c}.F}{\mathit{final}((\mathsf{aut}_\circ, 3, \_, \mathsf{elem}_\circ))}}{\mathit{final}((|{:}_\circ, 5, (\mathsf{aut}_\circ, 3, \_, \mathsf{elem}_\circ)))}$$

3.8 Quantified Synchronization

The quantified synchronization ASTD is the most convenient addition, compared to statecharts. It allows for the modeling of an arbitrary number of instances of an ASTD which are executing in parallel, synchronizing on events

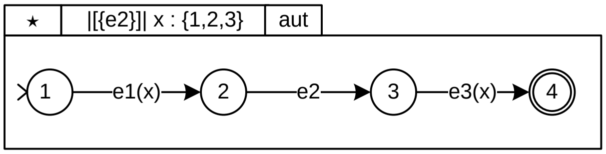

Figure 12. A closure over a quantified synchronization ASTD

from Δ. For IS modeling, it allows one to concisely and explicitly represent the behavior of each instances of an entity type or an association. In Harel's first paper on statecharts [14], this idea of quantification was mentioned as *parameterized states*. However, it has never been implemented in tools supporting statecharts, like Statemate [15]. Indeed, the main difficulty of this feature is in its implementation and automatic code generation. We have identified cases, which are frequently occurring in most IS specifications patterns, where we could generate efficient code that can deal with parameterized quantification. More discussion about this issue is provided in Section 5.

To illustrate the basic behavior, Figure 12 provides a simple quantified synchronization ASTD, nested in a closure. It denotes three concurrent instances of automaton a, i.e., one for each value of $x \in \{1, 2, 3\}$. These three automata synchronize on e2 (which is why e2 has no parameter). Hence, events e1(1), e1(2) and e1(3) can be received in any order. Once they have all been received, the three automata can synchronized on e2: the three automata execute e2 at the same time; from the view-point of the environment, a single event has been submitted. After e2, events e3(1), e3(2) and e3(3) can be received in any order. The quantification is in a final state when all its component automata are in a final state. Hence, a new iteration on the quantification can start only when all e3(x) have been received.

Figure 13 illustrates a more realistic and complex example, with two nested quantified synchronization ASTDs. It describes the invoicing of orders. The outer quantification includes a closure on an order automaton. The quantification on x allows to create any number of independent orders. Each order include its own quantification on its items. We require that when an order is invoiced, all its items are frozen and can't be modified, added or deleted, until the invoice is cancelled. This expressed by a synchronization on events invoiceOrder and cancelInvoice. Hence, when invoiceOrder(x) is received, all items of order x are synchronized and move to state 5 (which means invoiced). If the invoice is cancelled, each item of the order goes back to its previous state, thanks to the history state. An order can be deleted at any time.

Formal Definition and Semantics. Let QSynchronization $\triangleq$ $\langle |[]|{:}, n, x, T, \Delta, b\rangle$ be the set of quantified synchronization ASTDs, where $\Delta \subseteq$ Label denotes a synchronization set of event labels and $b \in$ ASTD is the quantified synchronized ASTD. The state of a quantified synchronization is of type $\langle |[]|{:}_\circ, f\rangle$ where $f \in T \to$ State is a function which associate a state to each value of T. Initial and final states are defined as follows.

$$
\begin{aligned}
\mathit{init}((|[]|{:}, n, x, T, \Delta, b)) &\triangleq (|[]|{:}_\circ, T \times \{\mathit{init}(b)\}) \\
\mathit{final}((|[]|{:}_\circ, f)) &\triangleq \forall\, v : T \cdot \mathit{final}(f(v))
\end{aligned}
$$

There are two inference rules: $|[]|{:}_1$ deals with events requiring no synchronization, while $|[]|{:}_2$ deals with the ones that do.

$$
|[]|{:}_1 \ \frac{\alpha(\sigma) \notin \Delta \qquad f(v) \xrightarrow{\sigma, (\![x:=v]\!) \lhd \Gamma}_b s'}{(|[]|{:}_\circ, f) \xrightarrow{\sigma,\Gamma} (|[]|{:}_\circ, f \mathbin{\lhd\!\!+} \{v \mapsto s'\})}
$$

$$
|[]|{:}_2 \ \frac{\alpha(\sigma) \in \Delta \qquad \forall\, v : T \cdot f(v) \xrightarrow{\sigma, (\![x:=v]\!) \lhd \Gamma}_b f'(v)}{(|[]|{:}_\circ, f) \xrightarrow{\sigma,\Gamma} (|[]|{:}_\circ, f')}
$$

3.9 Guard

A guard ASTD guards the execution of its component ASTD using a predicate. The first event received must satisfy the guard predicate. Once the guard has been satisfied by the first event, the component ASTD execute the subsequent events without further constraints from its enclosing guard ASTD. The predicate may refer to variables whose scope include the guard; in the context of IS specification, the guard could also refer to attributes of entities and associations, similarly to guards in process expressions of the EB^3 method [13].

The guard ASTD is a generalization of the guard specified on an automaton transition. It is especially useful when the component ASTD is a complex structure, avoiding the duplication of the guard predicate on all the possible first transitions of that structure.

An example. Figure 14 provides an example of a guard ASTD named c, which is included in the scope of a choice ASTD b, itself included in a Kleene closure ASTD a. The innermost component is the automaton d. If event e1(v) is received, then predicate $x > 0[(\![x := v]\!)]$, which reduces to $v > 0$ after applying the substitution, must hold for the event to be accepted; otherwise, it is rejected and ignored by the ASTD. If , event e1(v) is accepted, e2(v) can be accepted to terminate the first iteration of the closure. A new iteration can then start, and the new value v' for x must again satisfy $x > 0$. Figure 15 provides another example of a guard ASTD named c, which is included in the scope of a closure ASTD b, itself included in a quantified interleave ASTD a. The innermost component is the automaton d. ASTD a can spawn (so to speak) two instances of automaton d, one for $x := 0$ and one for $x := 2$. The instances for

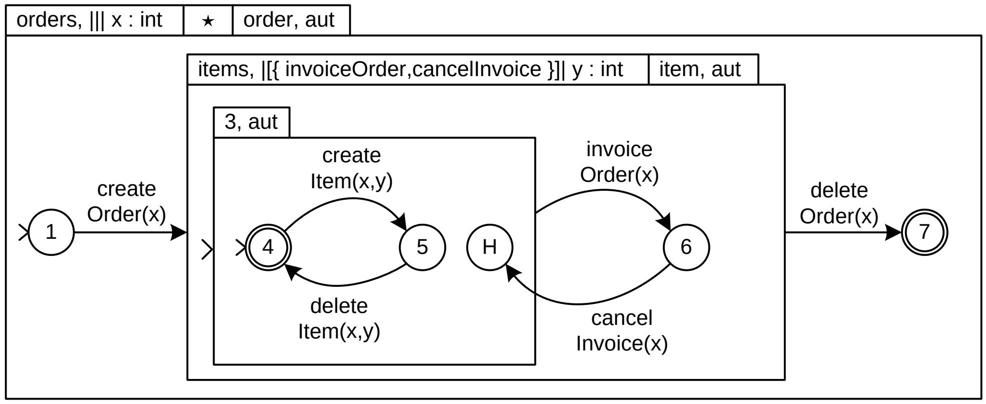

Figure 13. Invoicing of orders using a double synchronization quantification

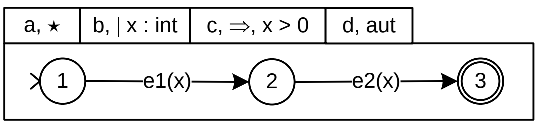

Figure 14. A guard ASTD nested in a closure on a choice

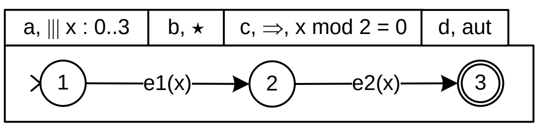

Figure 15. A guard ASTD nested in a quantified interleaved closure

$x \in \{1, 3\}$ cannot start, since they do not satisfy the guard. The initial state of a is a final state, since the body of the quantified interleave is a closure, whose initial state is also a final state, by definition of closure. ASTD a is also in a final state when the last two events received are e2(0) and e2(2).

Formal Definition and Semantics. Let Guard $\triangleq \langle \Rightarrow, n, g, b \rangle$ be the set of guard ASTDs, where $g \in$ Predicate is the guard predicate and $b \in$ ASTD is the guarded ASTD. The type of a guard state is $\langle \Rightarrow_\circ, started, s \rangle$ where $started \in$ Boolean and $s \in$ State. Symbol $started$ denotes whether the guard has been satisfied. It is set to false in the initial state and then set to true when the guard has been satisfied by the first transition. A guard ASTD is in a final state if i) it is not started, the guard predicate holds and the the initial state of its component ASTD is in a final state, or ii) it is started, and its component ASTD is in a final state. Here are the formal definitions of the initial state and the final states.

$$
\begin{aligned}
init((\Rightarrow, n, g, b)) &\triangleq (\Rightarrow_\circ, \mathsf{false}, init(b)) \\
final((\Rightarrow_\circ, \mathsf{false}, init(b))) &\triangleq g \wedge final(init(b)) \\
final((\Rightarrow_\circ, \mathsf{true}, s)) &\triangleq final(s)
\end{aligned}
$$

We need two rules of inference. The first one deals with the first transition and the satisfaction of the guard predicate. The second one deals with subsequent transitions.

$$
\Rightarrow_1 \frac{g[\Gamma] \qquad init(b) \xrightarrow{\sigma,\Gamma}_b s'}{(\Rightarrow_\circ, \mathsf{false}, init(b)) \xrightarrow{\sigma,\Gamma} (\Rightarrow_\circ, \mathsf{true}, s')}
$$

$$
\Rightarrow_2 \frac{s \xrightarrow{\sigma,\Gamma}_b s'}{(\Rightarrow_\circ, \mathsf{true}, s) \xrightarrow{\sigma,\Gamma} (\Rightarrow_\circ, \mathsf{true}, s')}
$$

3.10 ASTD Call

It is possible to call an ASTD which is defined in another diagram. A call is graphically represented by the called ASTD name and its actual parameter values. Calls can be recursive. Formally, let ASTDCall $\triangleq \langle \mathsf{cal}, n, P(\vec{v}) \rangle$ be the set of ASTD calls, where P is a reference to an ASTD definition $P(\vec{x} : \vec{T}) \triangleq b$ and, for each $v_i \in \vec{v}$, we have $v_i \in T_i$. The type of an ASTD call state is $\langle \mathsf{cal}_\circ, [\perp \mid s] \rangle$, where $\perp$ denotes that the call hasn't been made yet and $s \in$ State is actual state of the called ASTD when the called has been made. The initial and final states are as follows.

$$
\begin{aligned}
init((\mathsf{cal}, n, P(\vec{v}))) &\triangleq (\mathsf{cal}_\circ, \perp) \\
final((\mathsf{cal}_\circ, \perp)) &\triangleq final(init(b))[\vec{x} := \vec{v}] \\
s \neq \perp \Rightarrow final((\mathsf{cal}_\circ, s)) &\triangleq final(s)[\vec{x} := \vec{v}]
\end{aligned}
$$

There are two rules of inference. Rule cal_1 deals with the initial call execution, while cal_2 deals with subsequent executions.

$$\mathsf{cal}_1 \; \frac{init(b) \xrightarrow{\sigma, ([\![\vec{x}:=\vec{v}]\!]) \lhd \Gamma}_b s'}{(\mathsf{cal}_\circ, \bot) \xrightarrow{\sigma,\Gamma} (\mathsf{cal}_\circ, s')}$$

$$\mathsf{cal}_2 \; \frac{s \xrightarrow{\sigma, ([\![\vec{x}:=\vec{v}]\!]) \lhd \Gamma}_b s'}{(\mathsf{cal}_\circ, s) \xrightarrow{\sigma,\Gamma} (\mathsf{cal}_\circ, s')}$$

4 Case Study

In this section, we illustrate ASTDs on a very simple but typical IS case study. A library system has to manage loans of books by members. A book is acquired by the library. It can be discarded, but only if it is not lent. A member must register at the library in order to borrow a book. He/she can leave the library membership only when all his/her loans are returned.

Figure 16 defines the main ASTD which is a parameterized synchronization (parallel composition $\|$) of two ASTDs associated with the entity types of the library system, that is member and book. These two ASTDs are quantified synchronizations (interleave $|||$) where the quantified variables mid and bid take their value on the set of all the objects of, respectively, entity member and entity book. The unique component of each of these two quantified synchronizations ASTDs is an ASTD call that refers to the ASTD definition of member (resp. book) described in Figure 17. Each of them is a simple automaton describing the life cycle of an object. They refer to the loan automaton (Figure 17) that describes the life cycle of a loan of a given book bid by a given member mid. In the book automaton, the intermediate state is a closure on a quantified choice, meaning that a book can be borrowed several times but by only one member at a time, whereas in the member automaton, the intermediate state is a quantified synchronization on a closure, meaning that a member can borrow several books at a time.

In ASTD main, the parallel composition ($\|$) denotes the conjunction of the ordering constraints of each entity type. It ensures that if a book is borrowed, it satisfies the ordering constraints of the book and the member, since book and member must synchronize on common events, which are the events of ASTD loan. We have left out usual constraints like imposing a loan limit for a member, or taking into account reservations for books. They could easily be added to the model, by adding a guard for the loan limit in ASTD loan. Reservations can be modeled by adding a reservation ASTD that would composed in parallel with loans.

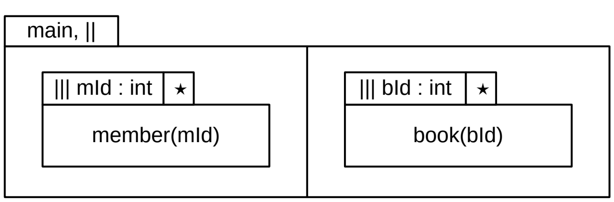

Figure 16. The main ASTD of the library case study

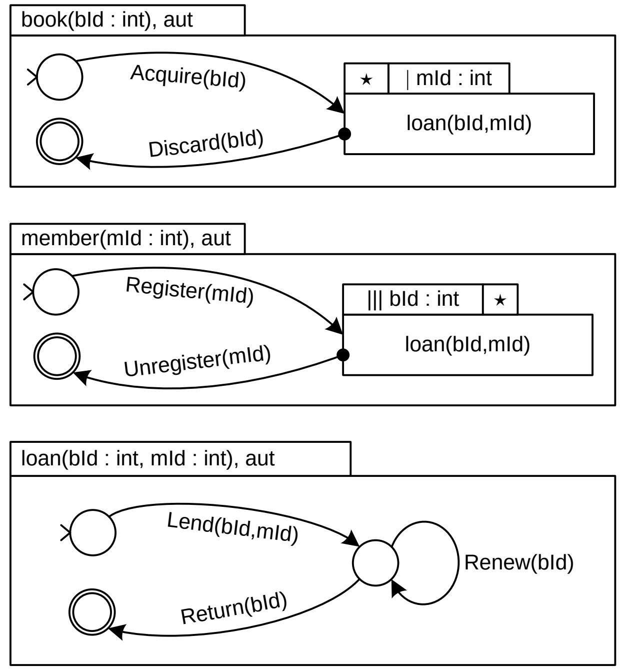

Figure 17. ASTDs describing a book, a member and a loan

5 Conclusion

We have introduced algebraic state transition diagrams, which allows for the combination of automata using traditional process algebra operators. Automaton states can themselves be ASTDs, supporting hierarchical decomposition of systems specifications, as in statecharts. Our motivation was the specification of IS, which require quantification operators to properly express the interaction between entities.

ASTDs provide a concise, yet comprehensive and formal, mechanism for specifying all the scenarios of an IS. Scenarios can be built incrementally and composed using process algebra operators. They make explicit the handling of entity instances, by using quantifications. Existing notations like statecharts are not convenient for capturing these aspects. Synchronization is also more convenient for IS modeling than statechart event broadcasting, according to our modeling experimentations. The syntax of ASTDs is quite simple, relying on well-known concepts. For the sake of simplicity, several features of statecharts are intentionally ignored, like entry and exit action for states, null transitions

(transitions without event labels), state predicates and static reactions. Our compositional semantics is also straightforward, using a simple labeled transition system. ASTD types and states are inductively defined and allow a free combination of all ASTD types.

We intend to develop an interpreter for ASTDs, which would be based on its operational semantics. We are confident that the techniques we have developed for our process algebra interpreter EB^{3}PAI [11, 12] can also be applied for ASTDs. Some patterns of synchronization quantification can be efficiently implemented by storing, for each value of the quantification variable, the state of the quantified ASTD. Typically, the quantification variable appears in each event of the quantified ASTD, so that its value can be extracted from the event and the state value can be retrieved efficiently, in $log(n)$ if a B-tree is used to store the mapping between the quantification values and the quantified ASTD state. Preliminary experimentations have shown that an ASTD interpreter can be faster and use less space than EB^{3}PAI.

References

[1] J.-R. Abrial. *The B-Book: Assigning Programs to Meanings*, Cambridge University Press, Cambridge, UK, 1996.

[2] M. Arbib. *Theories of abstract automata*. Prentice-Hall, 1969.

[3] J. A. Bergstra, J. W. Klop. Process Algebra for Synchronous Communication, *Information and Control* 60 (1):109–137, 1984.

[4] T. Bolognesi, E. Brinksma. Introduction to the ISO Specification Language LOTOS, *Computer Networks and ISDN Systems* 14 (1):25–59, 1987.

[5] Y. Bontemps, G. Saval, P. Heymans and P.-Y. Schobbens. From Interaction Diagrams to State Machines: Moving to Class-Level. In *AFADL 2006*, Paris, France, March 2006. ENST Technical Reports.

[6] E. Boerger, R. Staerk.: *Abstract State Machines: A Method for High-Level System Design and Analysis*. Springer-Verlag 2003.

[7] M. Broy and M. Wirsing. Algebraic State Machines. In *AMAST 2000*, LNCS 1816, 89–118, Springer-Verlag, 2000.

[8] W. Damn and D. Harel. LSCs: Breathing Life into Message Sequence Charts. *Formal Methods in System Design* 19(1):45–80 July 2001.

[9] J. Desharnais, M. Frappier and A. Mili. State transition diagrams. in *Handbook on Architectures of Information Systems*, 2nd edition, P. Bernus, K. Mertins, G. Schmidt, eds., 153–172, Springer-Verlag, 2006.

[10] J. Desharnais, M. Frappier, R. Khédri, A. Mili. Integration of Sequential Scenarios. *IEEE Transactions on Software Engineering*, 24(9):695–708 September 1998.

[11] B. Fraikin, M. Frappier. Efficient Symbolic Execution of Large Quantifications in a Process Algebra, in 9^{th} Int. Conf. on Formal Engineering Methods (ICFEM 2007), LNCS 4789, 327–344, Springer-Verlag, 2007.

[12] B. Fraikin, M. Frappier. Efficient Symbolic Execution of Process Expressions, submitted to *Science of Computer Programming*.

[13] M. Frappier and R. St-Denis. EB^3: an entity-based black-box specification method for information systems. *Software and Systems Modeling*, 2(2):134–149, July 2003.

[14] D. Harel. A visual formalism for complex systems. *Science of Computer Programming* 8, 231–274, 1987.

[15] D. Harel and A. Naamad. The STATEMATE Semantics of Statecharts. *ACM Trans. on Soft. Eng. and Meth.*, 5(4):293–333, October 1996.

[16] C. A. R. Hoare. *Communicating Sequential Processes*, Prentice-Hall, 1985.

[17] G. Lüttgen, M. von der Beeck and R. Cleaveland. Statecharts via Process Algebra. In *CONCUR'99*, LNCS 1664, 399–414, Springer-Verlag, 1999.

[18] F. Maraninchi. Argonaute: Graphical Description, Semantics and Verification of Reactive Systems by Using a Process Algebra. In *Automatic Verification Methods for Finite State Systems*, LNCS 407, 38–53, Springer-Verlag, 1989.

[19] R. Milner. *Communication and Concurrency*, International Series in Computer Science, Prentice Hall, 1989.

[20] Object Management Group. OMG Unified Modeling Language V2.1.2, http://www.omg.org.

[21] B. A. W. Roscoe. *The Theory and Practice of Concurrency*, amended 2005, 3rd Edition. Prentice Hall, 1998.

[22] M. Wirsing. Algebraic Specification. in *Handbook of Theoretical Computer Science, Vol. B*, 675–788, North Holland, 1990.

[23] J. Woodcock, J. Davies. *Using Z, Specification, Refinement and Proof*, Prentice Hall, 1996.

[24] J. Woodcock, A. Cavalcanti. The Semantics of Circus, in *ZB 2002: Formal Specification and Development in Z and B*, LNCS 2272, 184–203, Springer-Verlag, 2002.