

Systematic translation rules from ASTD to Event-B

Jérémy Milhau^{1,2}, Marc Frappier¹, Frédéric Gervais², and Régine Laleau²

¹ GRIL, Département Informatique, Université de Sherbrooke, 2500 boulevard université, Sherbrooke J1K 2R1, Québec, Canada
{Jeremy.Milhau,Marc.Frappier}@USherbrooke.ca

² Université Paris-Est, LACL, IUT Sénart Fontainebleau, Département Informatique, Route Hurtaut, 77300 Fontainebleau, France
{Frederic.Gervais,Laleau}@u-pec.fr

Abstract. This article presents a set of translation rules to generate Event-B machines from process-algebra based specification languages such as ASTD. Illustrated by a case study, it details the rules and the process of the translation. The ultimate goal of this systematic translation is to take advantage of Rodin, the Event-B platform to perform proofs, animation and model-checking over the translated specification.

Keywords: Process algebra, Translation rules, Systematic translation, ASTD, Event-B.

1 Introduction

Information Systems (IS) are taking an increasingly important place in today's organizations. As computer programs connected to databases and other systems, they induce increasing costs for their development. Indeed, with the importance of the Internet and their high computer market penetration, IS have become the de-facto standard for managing most of the aspects of a company strategy. In the context of IS, formal methods can help improving the reliability, security and coherence of the system and its specification. The APIS (Automated Production of Information system) project [7] offers a way to specify and generate code, graphical user interfaces, databases, transactions and error messages of such systems, by using a process algebra-based specification language. However, process algebra, despite their formal aspect, are not as easily understandable as semi-formal graphical notations, such as UML [13]. In order to address this issue, a formal notation combining graphical elements and process algebra was introduced: Algebraic State Transition Diagrams (ASTD) [8]. Using ASTD, one can specify the behavior of an IS. The interpreter *iASTD* [15] can efficiently execute ASTD specifications. However, there is no tool allowing proof of invariants or property check over an ASTD specification. This paper aims to define systematic translation rules from an ASTD specification to Event-B [2] in order to

model check or prove properties using tools of the RODIN platform [3]. Moreover, translation results will allow to bridge other process algebras (like EB³ [6] or CSP [11]) with Event-B as they share a similar semantics with ASTD. Event-B is first introduced to the reader in Section 2. An overview of ASTD and a case study will be then presented. This case study will help readers unfamiliar with ASTD to discover the formalism in Section 3. The Event-B machine resulting from translation rules applied to this case study will be described as well as rules and relevant steps of translation in Section 4. Finally, future work and evolution perspectives will be presented.

2 Event-B Background

Event-B [2] is an evolution of the B method [1] allowing to model discrete systems using a formal mathematical notation. The modeling process usually follows several refinement steps, starting from an abstract model to a more concrete one in the next step. Event-B specifications are built using two elements: *context* and *machine*. A *context* describes the static part of an Event-B specification. It consists of declarations of *constants* and *sets*. *Axioms*, which describe types and properties of constants and sets, are also included in the context. A *machine* is the dynamic part of an Event-B specification. It has a state consisting of several *variables* that are first initialized. Then *events* can be executed to modify the state. An event can be executed if it is enabled, *i.e.* all the conditions prior to its execution hold. These conditions are named *guards*. Among all enabled events, only one is executed. In this case, substitutions, called *actions*, are applied over variables. All actions are applied simultaneously, meaning that an event is atomic. The state resulting from the execution of the event is the new state of the machine, enabling and disabling events. Alongside the execution of events, *invariants* must hold. An invariant is a property of the system written using a first-order predicate on the state variables. In order to ensure that invariants hold, *proofs* are performed over the specification.

3 ASTD Background

ASTD is a graphical notation linked to a formal semantics allowing to specify systems such as IS. An ASTD defines a set of traces of actions accepted by the system. ASTD actions correspond to events in Event-B. Event-B actions and substitutions, as they modify the state of an Event-B machine, can be binded to the change of state in ASTD. The ASTD notation is based on operators from the EB³ [9] method and was introduced as an extension of Harel’s Statecharts [10]. An ASTD is built from transitions, denoting action labels and parameters, and states that can be elementary (as in automata) or ASTD themselves. Each ASTD has a type associated to a formal semantics. This type can be automata, sequence, choice, Kleene closure, synchronization over a set of action labels, choice or interleaving quantification, guard and ASTD call. One of ASTD most important features is to allow parametrized instances and quantifications, aspects missing

from original Statecharts. An ASTD can also refer to attributes, which are defined as recursive functions on traces accepted by the ASTD, as in the EB³ method. Such a recursive function compares the last action of the trace and maps each possible action to a value of the attribute it is defining. Computing this value may imply to call the function again on the remaining of the trace.

3.1 ASTD Operators

Several operators, or ASTD types, are used to specify an IS. We detail them in the following paragraphs. Operators will be further illustrated in Section 3.2 with the introduction of a case study.

Automata In an ASTD specification, one can describe a system using hierarchical states automata with guarded transitions. Each automata state is either elementary or another ASTD of whichever type. Transitions can be on states of the same depth, or go up or down of one level of depth. A transition decorated by a bullet (\bullet) is called a final transition. A final transition is enabled when the source state is final. As in Statecharts, an history state allows the current state of an automata ASTD to be saved before leaving it in order to reuse it later.

Sequence A sequence is applied to two ASTD. It implies that the left hand side ASTD will be executed and will reach a final state before the right hand side ASTD can start. There is no immediate equivalent of this operator in Harel's Statecharts, but its behavior can be reproduced with guards and final transitions. A sequence ASTD is noted with a double arrow \Rightarrow .

Choice A choice, noted $|$ allows the execution of only one of its operands, like a choice in regular expressions or in process algebras. The choice of the ASTD to execute is made on the first action executed. After the execution of the first action, the chosen ASTD is kept until it terminates its execution. If both operands of a choice ASTD can execute the first action, then a nondeterministic choice is made between the two ASTD. The behavior of a choice ASTD can be modeled in Statecharts using internal transition from an initial state, in a similar way to automata theory with ϵ transitions.

Kleene Closure As in regular expressions, a Kleene closure ASTD noted $*$ allows its operand to be executed zero, one or several times. When the state of its operand is final, a new iteration can start. There is no similar operator in Statecharts, but the same behavior can be reproduced with guards and transitions.

Synchronization Over a Set of Action Labels As the name suggests, this operator allows the definition of a set of actions that both operands must execute at the same time. It is similar to Roscoe's CSP parallel operator \parallel_x . There are some similarities with AND states of Statecharts and synchronization ASTD. A synchronization over the set of actions Δ is noted $||[\Delta]$. We derive two often used

operators from synchronization : interleaving, noted \parallel , is the synchronization over an empty set ; parallel, noted \parallel , synchronizes ASTD over the set of common actions of its operands, like Hoare’s CSP \parallel .

Quantified Interleaving A quantified interleaving models the behavior of a set of concurrent ASTD. It sets up a quantification set that will define the number of instances that can be executed and a variable that can take a value inside the quantification set. Each instance of the quantification is linked to a single value, two different instances have two different values. This feature lacks in Statecharts, as we have to express distinctly each instance behavior, but was proposed as an extension and named “parametrized-and” state by Harel. A quantified interleaving of variable x over the set T is noted $\parallel x : T$.

Quantified Choice A quantified choice, noted $| x : T$, lets model that only one instance inside a set will be executed. Once the choice is made, no more instances can be executed. As in quantified synchronization, the instance is linked to one value of a variable in the quantification set. An extension of Statecharts named a similar feature “parametrized-or” state.

Guard Usually, guards are applied to transitions. With the guard ASTD, one can forbid the execution of an entire ASTD until a condition holds. The predicate of a guard can use variables from quantifications and attributes. A predicate $P(x)$ guarding an ASTD is noted $\implies P(x)$

ASTD call An ASTD call simply links to other parts of the specification using the name of another ASTD. The same ASTD can be called several times, in different locations of the specification. It allows the designer to reuse ASTD in the same specification and helps synchronize processes. An ASTD call is made by writing the name of the ASTD called and its parameters (if any).

3.2 An ASTD Case Study

In order to present features and expressiveness of the ASTD notation to the reader, Fig. 1 introduces the case study that will be used throughout this paper. This ASTD models an information system designed to manage complaints of customers in a company. In this system, each complaint is issued from a customer relatively to a department. This example is inspired from [19]. The **main** ASTD, whose type is a synchronization over common actions, describes the system as a parallel execution of interleaved customers and departments processes. The IS lifecycle of a given customer is described by the parametrized ASTD **customer** (u), the same applies for the description of the company departments in the ASTD **department** (d). In the initial state, a customer or a department must be created. Then complaints regarding these entities can be issued. This is described using an ASTD call. The final transition means that the event can be executed if the source state is final. In our case, in order to delete a customer or a department, any related complaints must be closed. Finally, the

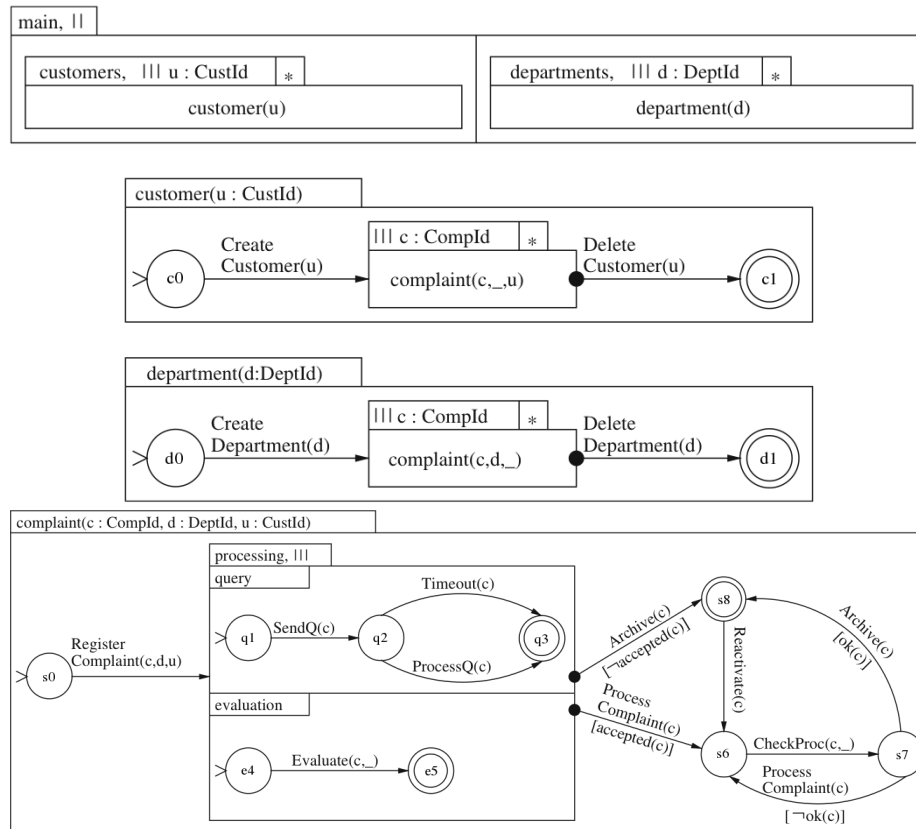


Fig. 1. An ASTD specification describing a complaint management system.

ASTD describing the checking and processing of a complaint c issued by customer c about department d is given by **complaint** (c, d, u). After registering a complaint in the system, it must be evaluated by the company and a questionnaire is sent to the customer in order to detail his/her complaint. The specification takes into account the possibility that the customer does not answer the questionnaire with the `Timeout(c)` event. Then, if the complaint is accepted and the questionnaire received or timed out, a check is performed. In case of refusal of the complaint, it is archived, but it can be reactivated later. The only final state is state $s8$, meaning that the complaint was archived (solved or not). In this specification, no attribute modification is performed. An ASTD only describes traces and has no consequences on updates to be performed against IS data, such as attributes that are stored in databases. However, an ASTD can access attribute values to use them in guards, as shown in both `Archive(c)` actions.

3.3 Motivations

ASTD are not the only way to specify IS behavior. The UML-B [18] method introduces a behavior specification in the form of a Statecharts. Using Statecharts, it is easy to describe an ordered sequence of actions whereas using B, it is easier to model interleaving events. A systematic translation of Statecharts into B machines is proposed by [17]. Compared to Statecharts, ASTD offer additional operators to combine ASTD in sequence, iteration, choice and synchronisation. When a UML-B specification models a system, it can only describe the life-cycle of a single instance of a class whereas ASTD specification models the behavior of all instances of all classes of the system. A new version of UML-B [14] introduces the possibility to refine class and Statecharts as part of the modeling process, and can translate it into Event-B. The UML-B approach can describe the evolution of entity attributes using B substitutions, a feature that ASTD lacks. csp2B [4] provides better proofs (on the B machine) and model checking (on the CSP side) tools than Statecharts but lacks the visual representation of the specification given by UML Statecharts. It is also limited to a subset of CSP specifications, where the quantified interleaving operator must not be nested. ASTD aims to be a compromise in both visual and synchronization aspects. On the other hand, ASTD lacks proofs and model checking allowed by the B side of UML-B and csp2B approaches. In order to answer this issue, a systematic translation of ASTD specifications into Event-B is proposed.

The choice between classical B and Event-B was made at an early stage by comparing tools and momentum of both methods. It appears that community efforts and tool development are currently focused on Event-B. Despite the fact that classical B offers some convenient notation such as IF / THEN statements or operation calls, Event-B appeared as a good compromise for our efforts. Classical B translation rules inspired by Event-B rules might be written.

4 Translation

Translation from ASTD to Event-B is achieved in several steps. Fig. 2 presents the architecture of the translation process. A context derived from ASTD operators introduces constants and sets needed to code their semantics. This context is the

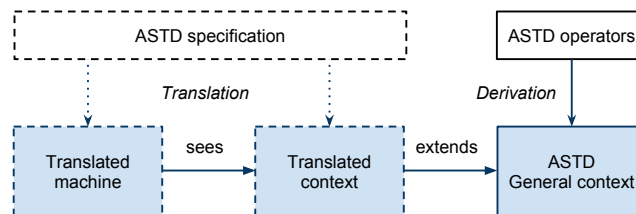


Fig. 2. The architecture resulting from the translation process.

Table 1. Event-B representation of ASTD states

ASTD state	State domain	Initial State
choice	$State \in \{ \mathbf{none}, \mathbf{first}, \mathbf{second} \}$	none
sequence	$State \in \{ \mathbf{left}, \mathbf{right} \}$	left
Kleene closure	$State \in \{ \mathbf{neverExecuted}, \mathbf{started} \}$	neverExecuted
synchronization	-	-
quantified choice	$State \in \{ \mathbf{notMade}, \mathbf{made} \} \leftrightarrow \text{QUANTIFICATIONSET}$	notMade $\mapsto 0$
quantified synch	$State \in \text{QUANTIFICATIONSET} \rightarrow \text{STATESET}$	initial for all
guard	$State \in \{ \mathbf{checked}, \mathbf{notChecked} \}$	notChecked
ASTD call	-	-

same in all translations and is described by Table 1. It codes elements from the semantics of all types of ASTD except automata, and is inspired of mathematical definition of ASTD semantics. Constants, sets and axioms defined in this context may be re-used in other part of the Event-B translation, hence this context is extended by a translation specific context. Automata states are translated into such a specific context since automata states depend on the ASTD specification to translate. For each ASTD, a variable and an invariant corresponding to its type are created. The invariant associates the variable to the set of values it can take, as defined in both contexts. In the following sections, we provide translation rules for each ASTD type, generating appropriate contexts and machines.

4.1 Automata

The first part of automata translation concerns the static part, the context. Several elements are introduced in the context: states, initial states, final states and transition functions.

States States from automata ASTD are represented as constants and grouped into state sets in order to facilitate later use. Even hierarchical states are represented by a constant.

Initial States Since an ASTD can be reset by the execution of a Kleene closure, initial states are defined as separate constants. They are also useful in the initialisation event of the machine generated in next step of our translation.

Final Predicates A final predicate is a function taking a state as argument and returning TRUE or FALSE depending if the state is final or not. The number of arguments depends on the type of the ASTD. This predicate is useful in the case of final transitions, sequences or Kleene closures, when transitions are activated if, and only if, a state is final. Hence, a final predicate is written for each ASTD type in the context common to all translations.

Transition Functions A transition function for each action label is generated. It takes as argument the current state of an automata ASTD and returns the resulting state. Transition functions are deterministic and partial.

The generated context for our case study defines 40 constants, 5 sets and 29 axioms. It is not presented here for the sake of conciseness. Then, for the dynamic part, for each distinct action label in the translated automata ASTD, a single event will be produced. If the action has a guard, a WHEN clause *i.e.* a guard, is generated. If the ASTD action has arguments (in the case of quantified variable for instance), an ANY clause is built accordingly and a guard specifying a type for the variable is added. Then a guard testing that the execution of the action is allowed *i.e.* the current state is in the domain of the transition function of the event. The modification of the state is applied by generating a THEN substitution.

Translation rules for automata ASTD are presented in Table 2. When a transition, an initial state or a final state is found, the first rule applies. In the case of a final transition, the second rule then applies. In the second pattern translation, the guard numbered **g1** of Table 2 is added to event e that was generated by applying first rule. In our case study, the second rule is applied for the `ProcessComplaint(c)` action. The guard added in this case is described by guard `grdAutomata`.

```
grdAutomata : isFinalProcessing(isFinalQuery(StateQuery(c))  $\mapsto$ 
isFinalEvaluate(StateEval(c))) =TRUE
```

Constants *isFinalX* and *StateX* refer to ASTD **X** in Fig. 1. An interleaved state is final if, and only if, both of its operand states are final. For this reason, guard `grdAutomata` checks if both states of **Query** and **Evaluation** ASTD are final. A pair (x, y) is noted $x \mapsto y$ in Event-B.

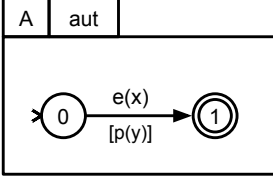
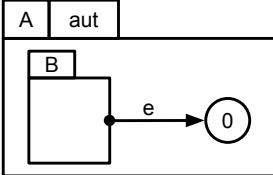
The action `CreateCustomer(u)` is translated into the event described below. *grd1* describes the set in which the parameter u can take its value. *grd2* verifies that a customer is in a state of the domain of transition function *TransCreateCustomer*. *act1* describes the state update for action `CreateCustomer(u)`: it only modifies the state of customer u according to the transition function *TransCreateCustomer*.

```
Event CreateCustomer  $\hat{=}$ 
  any
     $u$ 
  where
    grd1 :  $u \in USERSET$ 
    grd2 :  $StateCustomer(u) \in dom(TransCreateCustomer)$ 
  then
    act1 :  $StateCustomer(u) := TransCreateCustomer(StateCustomer(u))$ 
  end
```

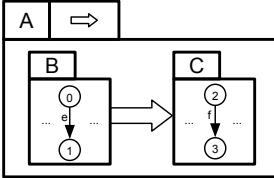
4.2 Sequence

Because of the number of possibilities to determine whether or not a sequence can switch from `left` state to `right` state, an extra event is introduced. This

Table 2. Automata ASTD to Event-B translation rules

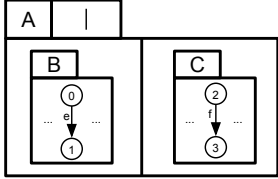
Automata ASTD pattern	Added to the context	Modifications on the machine
	<p>SETS</p> <p>StatesA</p> <p>CONSTANTS</p> <p>s0, s1 initA, isFinalA, TransE</p> <p>AXIOMS</p> <p>ax1 : $partition(StatesA, \{s0\}, \{s1\})$ ax2 : $initA = s0$ ax3 : $isFinalA = \{s0 \mapsto FALSE, s1 \mapsto TRUE\}$ ax4 : $TransE = \{s0 \mapsto s1\}$</p>	<p>Event $e \hat{=}$</p> <p>any x</p> <p>where g1 : $x \in XSET$ g2 : $P(y)$ g3 : $StateA \in dom(TransE)$</p> <p>then a1 : $StateA := TransE(StateA)$</p> <p>end</p>
	<p>CONSTANTS</p> <p>isFinalB</p> <p>AXIOMS</p> <p>ax1 : $isFinalB = \dots$ // Depends on B type</p>	<p>Event $e \hat{=}$</p> <p>where g1 : $isFinalB(StateB) = TRUE$...</p>

event is similar to an internal event of the IS and will verify that all the conditions for the switch from left to right side to happen holds and then change the state of the sequence. For example, if an ASTD named **A** is a sequence of ASTD **B** and **C**, the generated event will be called *switchSequenceA*. Then, in order to ensure that the current state allows the execution of every events of ASTD **B** and **C**, a guard is added to each event of **B** and **C** to check if the state of ASTD **A** is *left* or *right* respectively. As for automata, a final predicate must be generated in the context for ASTD **B** state. Translation rule is described in the following table.

Sequence ASTD pattern	Modifications on the machine
	<p>Event $switchSequenceA \hat{=}$</p> <p>where g1 : $isFinalB(StateB) = TRUE$</p> <p>then a1 : $StateA := right$</p> <p>end</p> <p>Event $e \hat{=}$</p> <p>where g2 : $StateA = left$...</p> <p>Event $f \hat{=}$</p> <p>where g3 : $StateA = right$...</p>

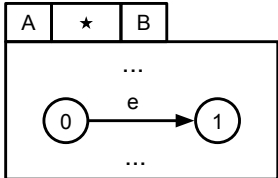
4.3 Choice

A choice ASTD can be in three states as described by the general ASTD context: **none** when the choice is not made yet, **first** or **second** depending of the side chosen. The translation rule for a choice ASTD is presented in the following table. If an ASTD named **A** is a choice between ASTD **B** and **C**, then a guard and an action are added to each event. Events from **B** will receive guard **g1** and action **a1**. A similar transformation of events from ASTD **C** is also needed with guard **g2** and action **a2**.

Choice ASTD pattern	Modifications on the machine
	<pre> Event e ≐ where g1 : StateA = first ∨ StateA = none ... then a1 : StateA := first ... Event f ≐ where g2 : StateA = second ∨ StateA = none ... then a2 : StateA := second ... </pre>

4.4 Kleene Closure

When an iteration of a Kleene closure ASTD is completed, its operand must be reset to initial state. For this reason, an additional event is generated. In the IS, this event is internal and hidden, in the ASTD specification, the semantics off Kleene operator handles the process, but in Event-B the reset must be described. This event will be activated when the its operand is final, and will reinitialize all sub-states in the hierarchy. The following table details the resulting Event-B machine.

Kleene ASTD pattern	Modifications on the machine
	<pre> Event lambdaA ≐ where g1 : isFinalB(StateB) = TRUE then a1 : StateB := initB And all sub states ... end Event e ≐ then a2 : StateA := started ... </pre>

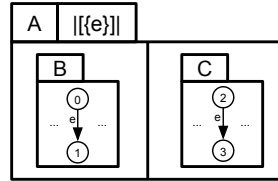
As presented for automata and sequence, a final predicate must be generated in the context for ASTD under the Kleene closure operator.

4.5 Synchronization Over a Set of Action Labels

For actions that are not synchronized, nothing is introduced or modified by the translation of synchronization ASTD. This is the case for interleaving ASTD and

action labels not common to both operands of the parallel operator. In the case of a synchronized action, guards from both operand must be put in conjunction, and substitutions applied conjointly.

Synchronization ASTD pattern Modifications on the machine



Event $e \hat{=}$
where
 gB : guardsfromBASTD
 gC : guardsfromCASTD
...
then
 $a1$: StateB := ...
 $a2$: StateC := ...
...

In our case study, the only synchronization ASTD is **main**. Common actions of both sides are only actions appearing in the ASTD **complaint** (c, d, u). For each one of the generated events of **complaint** (c, d, u), the guards **readyInCustomer** and **readyInDepartment** must hold. cc and dc states correspond to states where the customer and the department respectively are in the complaint quantified interleaving ASTD.

readyInCustomer : $StateCustomer(AssociationCustomer(c)) = cc$

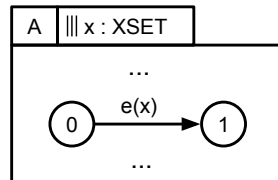
readyInDepartment : $StateDepartment(AssociationDepartment(c)) = dc$

Theses guards check that the customer associated to the complaint c is in the state allowing him to complain *i.e.* created and not deleted, and if the department associated to the complaint c exists in the IS.

4.6 Quantified Interleaving

The quantified interleaving does not introduce additional constraints to events. The following table shows how variables induced by quantified interleaving are handled in events.

Quantified interleaving ASTD pattern Modifications on the machine



Event $e \hat{=}$
any
 x
where
 $g1$: $x \in XSET$
 $g2$: $StateA(x) = \dots$
...
then
 $a1$: $StateA(x) := \dots$
...

Entities and associations patterns are common in EB³ and ASTD as mentioned in [9]. Such pattern are expressed using interleaving quantifications. In order to code in Event-B the association between several entities, a table variable must register their link. In our case study, we can see that a 1-n association between a customer and a complaint is created. When a complaint is created, an unique customer u is linked to the complaint c . The same applies to the department associated to the complaint. An Event-B variable is created in order to save the link between a complaint and a customer (respectively a department) and is updated whenever a complaint is registered in the system.

4.7 Quantified Choice

Similarly to the choice operator, the quantified choice implies that for all events using it, a check is performed about whether the choice was made or not. In the case of an action labeled e and taking x as a parameter, where x is the variable of a quantified choice ASTD named \mathbf{A} then the guard $\mathbf{g2}$ described in the following table must hold. The substitution $\mathbf{a1}$ must also be executed in case this is the first call of an action with this quantified variable. All the events of ASTD \mathbf{A} will be modified to include this guard and substitution.

Quantified choice ASTD pattern	Modifications on the machine
	<pre> Event $e \hat{=}$ any x where $\mathbf{g1} : x \in XSET$ $\mathbf{g2} : StateA = (qNone \mapsto 0) \vee StateA = (qSome \mapsto x)$... then $\mathbf{a1} : StateA := (qSome \mapsto x)$... </pre>

4.8 Guard

There are two cases for guard state: the guard was checked and held when we executed an event ; the guard did not hold, and no event was executed. These cases are handled with guard $\mathbf{g1}$ and substitution $\mathbf{a1}$ for a guard ASTD named \mathbf{A} guarded with predicate $P(x)$. All the events of ASTD \mathbf{A} will be modified to include this guard and substitution.

Guard ASTD pattern	Modifications on the machine
	<pre> Event $e \hat{=}$ any x where $\mathbf{g1} : StateA = checked \vee (StateA = notChecked \wedge P(x))$... then $\mathbf{a1} : StateA := checked$... </pre>

4.9 Process call

An ASTD that calls other ASTD does not need any constraint over its actions in Event-B. The translation will be achieved as if the entire called ASTD was substituted for the ASTD call. We do not deal with recursive ASTD calls yet.

When the translation process is completed, we can now access all the tools offered by Rodin to animate, model check and prove elements of the translated ASTD specification.

5 Animation and Model Checking of the Case Study

The final generated system, a context and a machine, translated from our case study represents 270 lines of Event-B, including 40 constants, 5 sets and 29 axioms for the static part and 7 variants, 7 invariants, 17 events (one for initialization, 13 representing ASTD actions and 3 internal events for Kleene Closure induced resets) representing 57 guards and 33 actions for the dynamic part. During the construction of translation rules, animation helped to correct rules, to improve the quality of translation rules and to factor contexts in order to separate static elements from machine. It was chosen to limit the size of quantification sets to three elements each. Only three departments, customers and complaints can be registered inside the system at any time. The screen capture was taken after the execution of 150 events and shows the state of variables of the machine. In order to informally verify the consistency of the Event-B machine with the initial ASTD specification, we generated a set of traces of events executed via the ProB animator. Then, for each trace, we removed the internal events introduced by the translation process such as `lambdaComplaint(c)`. Then we interpreted the initial ASTD specification with *i*ASTD and executed the traces. We could not find a trace of events that could not be interpreted by *i*ASTD. A more formal proof of the consistency of the translation must be performed, but first results are encouraging. Formal proof of translation rules is work in progress, and will be based on simulation.

Regarding the Event-B machine, 86 proof obligations were generated and 62 were automatically proved. The 24 remaining are proved manually and involved functional and set operators that are known for not being proved automatically. The manual proofs raised no specific difficulty. This Event-B specification was model checked for deadlocks and invariant violations using the consistency checking feature of ProB. More than 111 500 nodes were visited and 226 000 transitions activated. No deadlock nor invariant violation were found. More invariant properties might be written in order to be proved. Since ASTD only focuses on event control and not on event effects on the IS, when an event is executed, there is no way to know only by looking at the ASTD specification how IS state will evolve. Hence, no invariant can be generated during the translation. But it could be interesting to express invariants on ASTD as it was done with Statecharts [16]. For instance we could add an invariant to ASTD **Department** saying that whenever transition `DeleteDepartment(d)` is active, no complaint about this department must be registered in the system.

6 Limitations, Conclusion and Future Work

We have presented a set of translation rules allowing generation of Event-B contexts and machines from ASTD specifications. The animation of the resulting machine using ProB [12] animator helped to find errors and to tweak translation rules. Kleene closure and sequence operators were the most tricky to translate since these operators defines the ordering of events and because they introduce additional events in order to code semantics of ASTD in Event-B. A formal proof of the translation rules will be performed in order to entrust the translation process.

Refinement is one of the most important features of Event-B modelling process. In our approach, this aspect is missing. Indeed, we are translating an ASTD specification modelling a concrete system. Because of that, there is no need to refine the Event-B machine resulting from the translation process. It would have been relevant to introduce refinement in the translation process if a similar notion existed in ASTD, but it is currently not the case. Proof is an important aspect of Event-B that our approach would like to take advantage of. Alongside with formal IS specification, we advocate writing security or functional properties during the modeling process. This way, properties can be checked against the system as soon as it is modeled. Expressing these properties as Event-B invariants and proving invariant preservation in the translated machine is an important step of IS specification validation. Another feature of Event-B we do not use is composition. This may be very useful for the translation of some ASTD operators such as synchronization. It could lead to a more modular approach of translation, in a way similar to ASTD.

It would be interesting to compare the machine resulting of the translation process with a hand-written Event-B specification for the same system. Indeed, we would like to know if the automatic prover can do the same job with the hand-written and the translated machine. This study is work in progress and may result in an evolution of translation rules. Another step that we currently work on is to implement an ASTD modeler as a Rodin plugin. Using benefits from The Eclipse Graphical Modeling Framework (GMF) [5], a graphical editor could be used to build complete IS specifications. One could interpret them using the *i*ASTD [15] interpreter and then translate them to Event-B on the fly in order to perform model checking or proofs. This integrated tool would allow a great flexibility and would combine advantages of process algebra's power of expression, graphical representation's ease of understanding and Event-B's tools for proving, checking and animating.

Acknowledgements : The authors would like to thank the anonymous referees for their insightful comments. This research is financed by ANR (France) as part of the SELKIS project (ANR-08-SEGI-018) and supported by NSERC (Canada).

References

1. Abrial, J.R.: The B-Book: Assigning Programs to Meanings. Cambridge University Press (1996)
2. Abrial, J.R.: Modeling in Event-B. Cambridge University Press (2010)
3. Abrial, J.R., Butler, M., Hallerstede, S., Voisin, L.: An open extensible tool environment for Event-B. *Lecture Notes in Computer Science* 4260, 588 (2006)
4. Butler, M.: csp2b: A practical approach to combining CSP and b. *Formal Aspects of Computing* 12(3), 182–198 (November 2000)
5. Eclipse Consortium: Eclipse graphical modeling framework (gmf), <http://www.eclipse.org/modeling/gmf/?project=gmf>
6. Fraikin, B., Frappier, M.: Efficient symbolic computation of process expressions. *Science of Computer Programming* (2009)
7. Fraikin, B., et al.: Synthesizing information systems: the APIS project. In: Rolland, C., Pastor, O., Cavarero, J.L. (eds.) *First International Conference on Research Challenges in Information Science (RCIS)*. p. 12. Ouarzazate, Morocco (Apr 2007)
8. Frappier, M., Gervais, F., Laleau, R., Fraikin, B., St-Denis, R.: Extending statecharts with process algebra operators. *Innovations in Systems and Software Engineering* 4(3), 285–292 (2008)
9. Frappier, M., St-Denis, R.: EB³: an entity-based black-box specification method for information systems. *Software and System Modeling* 2(2), 134–149 (2003)
10. Harel, D.: Statecharts: A visual formalism for complex systems. *Science of computer programming* 8(3), 231–274 (1987)
11. Hoare, C.A.R.: *CSP—Communicating Sequential Processes*. Prentice Hall (1985)
12. Leuschel, M., Butler, M.: ProB: A model checker for b. In: *FME 2003: Formal Methods*. *Lecture Notes in Computer Science*, vol. 2805, pp. 855–874. Springer Berlin / Heidelberg (2003)
13. Rumbaugh, J., Jacobson, I., Booch, G.: *The unified modeling language*. University Video Communications (1996)
14. Said, M.Y., Butler, M., Snook, C.: Language and tool support for class and state machine refinement in UML-B. In: *FM 2009: Formal Methods*. *Lecture Notes in Computer Science*, vol. 5850, pp. 579–595. Springer Berlin / Heidelberg (2009)
15. Salabert, K., Milhau, J., et al.: iASTD : un interpréteur pour les ASTD. In: *Atelier Approches Formelles dans l’Assistance au Développement de Logiciels (AFADL 2010)*. pp. 3–6. Actes AFADL, Poitiers, France (9-11 June 2010)
16. Sekerinski, E.: Verifying Statecharts with State Invariants. In: *13th IEEE International Conference on Engineering of Complex Computer Systems*. pp. 7–14. IEEE (2008)
17. Sekerinski, E., Zurob, R.: Translating statecharts to b. In: *Integrated Formal Methods*. *Lecture Notes in Computer Science*, vol. 2335, pp. 128–144. Springer Berlin / Heidelberg (2002)
18. Snook, C., Butler, M.: UML-B: Formal modeling and design aided by UML. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 15(1), 122 (2006)
19. Van Der Aalst, W.M.P.: The application of Petri nets to workflow management. *The Journal of Circuits, Systems and Computers* 8(1), 21–66 (1998)