DOI 10.1007/s00165-013-0286-3
BCS © 2013
Formal Aspects of Computing (2014) 26: 919–941

Formal Aspects of Computing

Refinement patterns for ASTDs

Marc Frappier[1], Frédéric Gervais[2], Régine Laleau[2] and Jérémy Milhau[2]

[1] GRIL, Département d'informatique, Université de Sherbrooke, 2500 Boulevard Université, Sherbrooke, QC, J1K 2R1, Canada
[2] Département Informatique, LACL, Université Paris-Est, IUT Sénart Fontainebleau, Route Hurtault, 77300, Fontainebleau, France

Abstract. This paper introduces three refinement patterns for algebraic state-transition diagrams (ASTDs): state refinement, transition refinement and loop-transition refinement. These refinement patterns are derived from practice in using ASTDs for specifying information systems and security policies in two industrial research projects. Two refinement relations used in these patterns are formally defined. For each pattern, proof obligations are proposed to ensure preservation of behaviour through refinement. The proposed refinement relations essentially consist in preserving scenarios by replacing abstract events with concrete events, or by introducing new events. Deadlocks cannot be introduced; divergence over new events is allowed in one of the refinement relation. We prove congruence-like properties for these three patterns, in order to show that they can be applied to a subpart of a specification while preserving global properties. These three refinement patterns are illustrated with a simple case study of a complaint management system.

Keywords: ASTD, Refinement, Patterns, Information systems

1. Introduction

Constructing a large specification of a system is a difficult and challenging task. Stepwise and iterative construction are now commonly used in software engineering for gradually tackling the complexity of a system. Patterns are extensively used in software design and programming [Cop03, GHJV94] to document recurring solutions and facilitate their application to new problems. Patterns are also used, somewhat less frequently, in software specifications for various purposes like specifying business processes [AtH12], information systems [FSD03] and temporal properties [BGPS12, DAC99]. Specification patterns, similar to design patterns, describes recurring specification structures. When a specification pattern is used to describe some part of a system, it may be useful to leave out some of the details of the specification, in order to tackle its complexity in several steps. Specification *refinement* patterns describe how to gradually introduce such details. Thus, specification refinement patterns are complementary to specification patterns. For instance, the work in [DvL96] supports stepwise and iterative construction of formal goals expressed in logic using formula decomposition. In this paper, we are interested in the stepwise construction of specifications represented by state-transition diagrams, more specifically using *algebraic state-transition diagrams* (ASTDs) [FGL+08]. The ASTDs notation combines the graphical power of a visual notation like hierarchical state diagrams *à la* Statecharts with process algebra operators, which streamlines the specification of entity life cycles in information systems (IS). In this paper, we propose three refinement patterns for ASTDs to facilitate the stepwise construction of ASTD specifications.

Correspondence and offprint requests to: M. Frappier, E-mail: Marc.Frappier@USherbrooke.ca

These patterns originates from several case studies constructed during two industrial research projects, SELKIS [MIL+11b] and EB^3 SEC [EJFG+10]. These two projects focused on the formal specification of IS and their security policies. In the EB^3 SEC project, one of our partners in the canadian banking industry provided us with case studies concerning brokers, customers and external financial systems, in order to manage investment portfolio and trade of financial products, like stocks or options. The SELKIS project deals with the medical domain, with two main case studies. The first one is focused on the data records from medical imaging devices in a French hospital. Previously designed for an internal use only, the IS has been updated in order to allow physicians to access these medical data records through web-based applications, even outside the hospital. The other one addresses availability and confidentiality issues for medical emergency units evolving in a great mountain range, like the Alps in that case. When specifying these case studies using ASTDs, we saw the benefit of using specification patterns similar to those described in [FGL+08], but we also noticed that it was easier to construct specifications in a stepwise manner, and we identified three recurring patterns for refining ASTD specification. These patterns are called *state refinement*, *transition refinement* and *loop-transition refinement*.

During these projects, we quickly realized that we also needed to formally analyze the relationship between a specification and its refinement, in order to understand what properties are preserved during refinement. Thus, we also propose in this paper a refinement relation which describes the connections between a specification and its refinement when these patterns are used. IS specifications are typically constructed by describing scenarios which, in the context of this paper, denote the order in which events are executed in a system. Our refinement relation preserves scenarios through each refinement step. It allows for the introduction of new events and the replacement of an abstract event by new events. This means that if a sequence of events is possible in the abstract specification, it is also possible in the more concrete specification, modulo the replacement of abstract events by more concrete events and the insertion of new events in the scenarios. Since refinement patterns are often applied to sub-parts of a specification, we also investigated the impact of refining a sub-part on the whole system. Thus, we provide theorems on congruence-like properties which state conditions under which scenarios of the global system are preserved when refining one of its sub-parts using the proposed refinement patterns. Since scenarios are preserved by our refinement relation when the conditions of these theorems hold, LTL and CTL properties proved on an abstract specification also hold on the more concrete specification, which is a major advantage for the stepwise construction of system specifications. Hence, system properties expressed in temporal logics can be preserved during the refinement process. This can help in reducing the model checking time of temporal properties, by introducing them at the appropriate step in the specification construction process.

The paper is organized as follows. Section 2 provides a brief overview of related work in specification refinement and specification refinement patterns for state-transition diagrams. Section 3 introduces the ASTD notation and describes its operational semantics in order to make this paper self-contained. Section 4 introduces the definition of refinement which is satisfied by the proposed refinement patterns. Section 5 introduces the proposed refinement patterns and their associated proof obligations. Section 6 provides proofs that the refinement patterns indeed satisfy the proposed definition of refinement, thus ensuring that the properties stated in the definition of refinement are indeed preserved by our patterns. Section 7 provides congruence-like properties for our refinement patterns, in order to study their impact on a specification when they are used on a sub-part of it. Section 8 compares our definition of refinement with two of the most common refinement relations for the stepwise construction of specifications, namely refinement in CSP and EVENT-B. Finally, Sect. 9 concludes the paper with an appraisal of this work and an outline of its possible extensions.

2. Related work

We first describe some of the main refinement relations used in formal methods, since we are targeting refinement patterns. We then describe related work on refinement patterns for state-transition diagrams.

There are numerous definitions of refinement [Gla90]. Model-based specification methods like ASM, B, VDM and Z include a notion of refinement which originates from sequential program development, where a component can be replaced by another without loss of observable behaviour. This refinement consists of weakening the precondition of operations and strengthening their postconditions; it is based on Dijkstra's notion of total correctness. It can be achieved by *algorithmic* refinement (*i.e.*, refining the description of an operation by providing an algorithm for its implementation) or by *data* refinement (changing the representation of the state space of the system by using more concrete data structures). B refinement [Abr96] is a typical example of this approach which has been used on several large industrial systems development. B refinement does not allow for the introduction of new events. Another closely related refinement approach, which originates from action systems, concurrent systems and distributed systems [BKS83, BKS88], allows for guards strengthening [BvW94] and nondeterminacy reduction. EVENT-B refinement [Abr10] is inspired from this approach: new events can be introduced and an abstract event can be split into several concrete events. Since guard strengthening can be used, the observable behaviour of the abstract system can be reduced during refinement. Refinement in event-based approaches like process algebras, of which CSP is a well know example, share some similarities with refinement in action systems. In CSP, the set of accepted traces can be reduced using trace refinement, nondeterminacy can be reduced using failure refinement, and divergences can be reduced using failure-divergence refinement [RHB98]. Hybrid approaches (i.e., both state-based and event-based) like CIRCUS [WC02] and CSP ‖ B [ST11], support both forms of refinement. Refinement based on Dijkstra's total correctness is often used, although not exclusively, for *implementing* a specification; refinement in event-based (or action-based) systems is often used for *constructing* a system specification, although again not exclusively. In some methods like CIRCUS, specification construction and implementation are seamlessly integrated.

Our notion of refinement is essentially targeted for system specification construction, especially for IS. It allows for the introduction of new events and for the replacement of abstract events by concrete events. It also enforces the preservation of behaviour. As a convention, when a specification A is refined by a specification B, we refer to A as the *abstract* specification and B as the *concrete* specification. In our notion of refinement, each trace of the abstract specification must be matched by a trace of the concrete specification, and vice-versa. Abstract traces are made more concrete by inserting new events, or by replacing an abstract event by concrete events. The ASTDs used in this paper are purely event based (they do not contain state variables like model-based notations); thus we do not use data refinement. Our approach is closer to EVENT-B refinement and CSP refinement, since we are targeting system specification construction. In Sect. 8, we provide a detailed comparison with both, to show the subtle, but important, differences between them.

We now turn our attention to refinement patterns for state-transition diagrams. In [Sch98], refinement rules are defined for a dialect of Statecharts called μ-charts that is composed of three syntactic elements: sequential automata, hiding and parallel composition. The refinement rules allow for adding and removing states and transitions, or introducing parallel composition. New events can be introduced, but existing events cannot be replaced by new (concrete) ones, as we propose in our approach. Input-output traces of the concrete μ-chart must be a subset of the abstract μ-chart, when restricting traces to abstract events. In contrast, our approach does not permit to remove traces by refinement. Our refinement proof obligations ensure that the concrete ASTD satisfies some critical properties such as deadlock-freeness, livelock-freeness and fairness, which is also similar to EVENT-B refinement. In [Sai56], the author describes the refinement of Statecharts using EVENT-B as the semantics. Since the final objective of this work is to translate UML diagrams (class and state diagrams) into EVENT-B, the proposed Statechart refinement is based on EVENT-B refinement. We will compare EVENT-B refinement with our refinement in Sect. 8.2.

In [MNB04], Statecharts are seen as coalgebras [Rut00]. In that theory, one can consider (bi)simulations between the different observations of Statecharts. The authors have proposed a definition of behaviour refinement adapted for Statecharts. They have also defined a set of refinement laws for transforming Statecharts. These laws operate on elementary components (for instance, adding a state, removing an unreachable state, etc.), while our refinement patterns allows for the replacement of an event or a state by an arbitrarily complex ASTD. The approach of [SK10] is quite similar, but adapted for hierarchical UML Statecharts, which is another variant of Statecharts. It uses a semantics which is very close to the UML standard. One interesting contribution is the introduction of

a *redefinable* state, which prevents some behaviours to be dropped by refinement. As in [MNB04], the refinement patterns are more concrete than ours.

In [CMP08], the authors discuss the proof obligations generated by colored Petri-net refinements. There are three ways to refine a colored Petri-net. (i) Type refinement is the refinement of the type of the tokens circulating in the net. There is no modification of the structure of the Petri-net with this refinement. (ii) Node refinement is a refinement of either a place or a transition. It replaces a place, respectively a transition, by a subnet bordered by places, respectively by transitions. (iii) Subnet refinement can introduce places, transitions, arcs or tokens. Two of our patterns are to some extent similar to the second Petri-net refinement: they are based on the replacement of a transition or an automaton state in an ASTD by a new ASTD.

3. The ASTD notation

The ASTD notation [FGL+08] is a graphical representation linked to a formal semantics designed to specify IS, although it is general enough to be used for other classes of systems. An ASTD is a state-transition diagram, but with an algebraic flavour, in the sense that ASTDs can be composed with operators. The ASTD notation was introduced as an extension of Harel's Statecharts [Har87], with the intent of composing hierarchical automata with process algebra operators like sequence, choice, Kleene closure, guard, synchronization, and quantification on choice and interleave. An elementary ASTD is a plain automata. As in Statecharts, automaton states can be ASTDs themselves, like OR and AND states in Statecharts. However, an automaton state can be of any ASTD subtype (*i.e.*, sequence, choice, Kleene closure, ...). One of the main important features of ASTDs is to allow parameterized instances and quantifications, aspects missing in Statecharts for easily specifying IS. This means that an ASTD can describe not only the behaviour of one instance of an entity, but also of all entities and relationships of the system. The operational semantics of the ASTD notation is formally described by inferences rules and axioms that allow one to prove that an action can be executed by constructing a proof tree. The reader is referred to [FGL+08, FGLFdf] for a complete description of these rules. In this paper, we use the labelled transition relation determined by these rules to define the proof obligations for the refinement patterns.

For the sake of concision, a *sum* type T is written as $T \triangleq \langle cons_1, T_1\rangle \mid \ldots \mid \langle cons_m, T_m\rangle$. Each $\langle cons_i, T_i\rangle$ is a Cartesian product where $cons_i$ denotes a sum tag and each $T_i = c_1, \ldots, c_n$ denotes the coordinate names of the Cartesian product. A tuple $t \in T_i$ is written $t = (cons_i, v_1, \ldots, v_n)$. We write $t.c_j$ to denote the value v_j of coordinate c_j of t. The type ASTD consists of the sum of the following subtypes:

$$\begin{aligned}\mathsf{ASTD} \;=\; & \mathsf{Automaton} \mid \mathsf{Sequence} \mid \mathsf{Guard} \mid \mathsf{Closure} \mid \mathsf{Choice} \mid \mathsf{Synchronization} \mid \\ & \mathsf{QChoice} \mid \mathsf{QSynchronization} \mid \mathsf{ASTDCall} \mid \mathsf{Elem}\end{aligned}$$

For each ASTD type T, we define a state type $T_\circ$, and define State as the sum of these types.

$$\begin{aligned}\mathsf{State} \;=\; & \mathsf{Automaton}_\circ \mid \mathsf{Sequence}_\circ \mid \mathsf{Guard}_\circ \mid \mathsf{Closure}_\circ \mid \mathsf{Choice}_\circ \mid \mathsf{Synchronization}_\circ \mid \\ & \mathsf{QChoice}_\circ \mid \mathsf{QSynchronization}_\circ \mid \mathsf{ASTDCall}_\circ \mid \mathsf{Elem}_\circ\end{aligned}$$

The semantics of an ASTD A is given by a transition relation $LTS(A) \subseteq \mathsf{State} \times \mathsf{Event} \times \mathsf{State}$ using a set of inference rules in the Plotkin style. Let $L(A)$ be the projection of $LTS(A)$ on State × State, and let $M(A) \triangleq dom(L(A)) \cup ran(L(A))$ be the states of $L(A)$. Let $I(S)$ be the identity function on a set S, let $s.U$ be the set of images of s by relation U, and let U^+ and U^* respectively denote the transitive and the reflexive-transitive closure of relation U. Function *init* : ASTD → State returns the initial state of ASTD A; it is inductively defined based on the ASTD subtypes. Function *final* : State × ASTD → Boolean returns *true* when a state is final; it is also inductively defined based on the subtypes of ASTD states. We denote by $F(A) = \{s \in \mathsf{State} \mid final(s, A)\}$.

Variables can be declared in ASTDs in two ways: (i) as parameters of an ASTD; these parameters are valued when an ASTD is called; (ii) as quantified variables introduced by a quantified choice or by a quantified synchronization. To deal with the valuation of variables for transition computation, we need the notion of an execution environment. An environment $\Gamma = (\!|x_1, \ldots, x_n := v_1, \ldots, v_n|\!)$ is a function which maps a variable x_i into a value v_i. When p is a formula or a term, $p[\Gamma]$ denotes the substitution of x_i by v_i in p. We write transitions with respect to Γ as $s \xrightarrow{\sigma,\Gamma}_{\mathsf{a}} s'$. We compute a transition of $LTS(A)$ starting from an empty environment ($(\!|\,|\!)$), using the following inference rule.

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

The terms *event* and *action* are often interchangeably used in the literature. In the context of this paper, an event is seen as the execution of an action.

In the rest of this section, we briefly describe the ASTD types which are used in the refinement patterns and in the case study illustrating these patterns.

Automaton ASTD**.** An ASTD automaton is similar to a classical 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. Let $\mathsf{Automaton} \triangleq \langle \mathsf{aut}, \Sigma, N, \nu, \delta, SF, DF, n_0 \rangle$ be the set of ASTDs of type automaton. We have the following typing constraints on the components of an automaton.

- Σ is the alphabet.
- N is the set of automaton state names.
- $\nu \in N \rightarrow \mathsf{ASTD}$ is a function that maps each state name to an ASTD.
- $\delta \subseteq \langle \eta, \sigma, \phi, \mathit{final}? \rangle$ is the transition relation, where:
  - η denotes the arrow. There are three types of arrows. Let $n_1, n_2, n_{1_\flat}, n_{2_\flat} \in N$ in the following: $\langle \mathsf{loc}, n_1, n_2 \rangle$ 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 $\nu(n_1) \in \mathsf{Automaton}$.
  - σ the event of the transition.
  - ϕ is the transition guard.
  - *final*? is a boolean denoting if the transition can be triggered only when its source state is final; a final transition (*i.e.*, when *final*? = true) is decorated with a "•" at its source; it is useful essentially when the source state is a compound ASTD.
- $SF \subseteq N$ denotes the names of shallow final states and $DF \subseteq N$ denotes the names of deep final states, with $DF \cap SF = \varnothing$.
- $n_0 \in N$ is the name of the initial state.

A state of an automaton is of type $\mathsf{Automaton}_\circ = \langle \mathsf{aut}_\circ, \mathit{name}, \mathit{hist}, \mathit{sub} \rangle$ where $\mathit{name} \in N$, $\mathit{hist} \in N \rightarrow \mathsf{State}$ is the history function to model history states of Statecharts, and $\mathit{sub} \in \mathsf{State}$ is the substate of the state. The history function hist stores the last visited substate of a state. Functions *final* and *init* are defined as follows for an automaton.

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

Figure 1 illustrates an Automaton ASTD named a1, with parameter x. It has two elementary states, 0 and 2, of type Elem, and a compound state 1, of type Automaton. We distinguish between the *name* of an automaton state, which belongs to N, and the *value* of an automaton state, which belongs to $\mathsf{Automaton}_\circ$. The *name* of the initial state of a1 is 0; the element of $\mathsf{Automaton}_\circ$ that denotes the initial state of a1 is $\mathit{init}(\mathsf{a1}) = (\mathsf{aut}_\circ, 0, h_0, \mathsf{Elem}_\circ)$, with $h_0 = \{0 \mapsto \mathsf{Elem}_\circ, 1 \mapsto (\mathsf{aut}_\circ, 3, \varnothing, \mathsf{Elem}_\circ), 2 \mapsto \mathsf{Elem}_\circ\}$. Since states 0, 2, 3 and 4 are elementary, they have no substate and hist is mapped to $\mathsf{Elem}_\circ$. Automaton a1(9) can execute the following transitions:

$$
\begin{aligned}
&(\mathsf{aut}_\circ, 0, h_0, \mathsf{Elem}_\circ) \xrightarrow{\mathsf{e1}(9)} \\
&(\mathsf{aut}_\circ, 1, h_0, (\mathsf{aut}_\circ, 3, h_2, \mathsf{Elem}_\circ)) \xrightarrow{\mathsf{e7}(9)} \\
&(\mathsf{aut}_\circ, 1, h_0, (\mathsf{aut}_\circ, 4, h_2, \mathsf{Elem}_\circ)) \xrightarrow{\mathsf{e4}} \\
&(\mathsf{aut}_\circ, 2, h_1, \mathsf{Elem}_\circ) \xrightarrow{\mathsf{e3}} \\
&(\mathsf{aut}_\circ, 1, h_1, (\mathsf{aut}_\circ, 4, h_2, \mathsf{Elem}_\circ))
\end{aligned}
$$

where $h_1 = h_0 \lhd \{1 \mapsto (\mathsf{aut}_\circ, 4, \varnothing, \mathsf{Elem}_\circ)\}$ and $h_2 = \{3 \mapsto \mathsf{Elem}_\circ, 4 \mapsto \mathsf{Elem}_\circ\}$, where $\lhd$ is the usual function override operator. Transition e4 is a final transition; it can be triggered only when state 1 is in a final state, *i.e.*, state 4. Transition e5 is a transition to an history state, which means that it returns to the last visited substate of 1, which is 4 in this case.

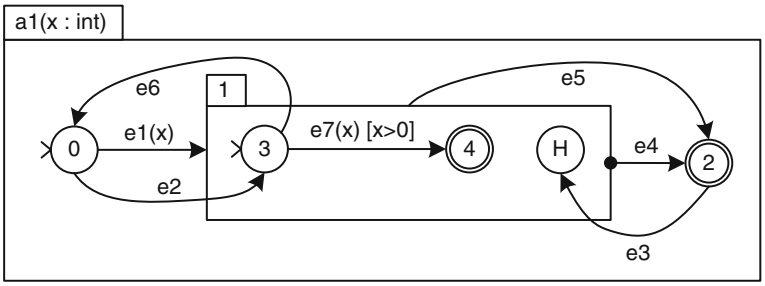

Fig. 1. A simple Automaton ASTD

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

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

This inference rule handles local transitions, taking into account the history function and the type of automaton states. The target state of the transition is the initial state of the destination state in δ: for an elementary state, it returns the elementary state itself; for other ASTDs, *init* returns the particular initial state of that structure. It includes a premiss denoted by Ψ and defined as follows (and also used in the other five rules): $\Psi \triangleq \big((\mathit{final}? \Rightarrow \mathit{final}(s)) \wedge g \wedge \sigma' = \sigma \wedge h' = h \lhd\!\!+ \{n_1 \mapsto s\}\big)[\Gamma]$. It provides that a transition denoted by *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 $\lhd\!\!+$, the override operator like in B [Abr96]. Rule aut_2 handles transitions to substates, in the particular case where the substate is not an history state:

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

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

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

In the case of shallow history, the target state is the *initial state* of the ASTD referenced by $h(n_2)$. Rule aut_4 handles transitions to a *deep* history state (noted H^*); in that case, the target state is the full state recorded in $h(n_2)$:

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

Rules aut_5 and aut_6 respectively handle transitions from a substate and transitions within a substate:

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

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

Kleene closure ASTD**.** 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. This behaviour is very common in IS. For instance, a typical pattern is the producer-modifier-consumer of an entity or an association (for a description of this pattern, see Sect. 5.1). The system can iterate an arbitrary number of times on the modifiers and then terminate with a consumer.

Formally, 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.

Operational semantics. 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.

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

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

Parameterized synchronization ASTD**.** The parameterized synchronization is similar to an AND-state in Statecharts, in that it allows two ASTDs to execute in parallel, but these two ASTDs 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 an interleave fashion. Thus, it is essentially the same behaviour as the parameterized synchronization found in process algebra like Lotos or Roscoe's version of CSP [RHB98]. 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 and it is written as $|||$.

Formally, the set of parameterized synchronization ASTDs is denoted by $\mathsf{Synchronization} \triangleq \langle |[]|, n, \Delta, l, r \rangle$, where $\Delta \subseteq \mathsf{Label}$ denotes a synchronization set of event labels and $l, r \in \mathsf{ASTD}$ are the synchronized ASTDs.

Operational semantics. There are three inference rules. Rules $|[]|_1$ and $|[]|_2$ respectively describe execution of events with no synchronization required on the left-hand side (LHS) and the right-hand side (RHS) of the synchronization ASTD. Expression $\alpha(a)$ denotes the labels of event appearing in ASTD a, including all its inner ASTDs.

$$|[]|_1 \frac{\alpha(\sigma) \notin \Delta \qquad s_l \xrightarrow{\sigma,\Gamma}_l s'_l}{(|[]|_\circ, s_l, s_r) \xrightarrow{\sigma,\Gamma} (|[]|_\circ, s'_l, s_r)} \qquad |[]|_2 \frac{\alpha(\sigma) \notin \Delta \qquad s_r \xrightarrow{\sigma,\Gamma}_r s'_r}{(|[]|_\circ, s_l, s_r) \xrightarrow{\sigma,\Gamma} (|[]|_\circ, s_l, s'_r)}$$

Rule $|[]|_3$ describes the synchronization between the LHS and the RHS:

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

Other ASTD **types.** The reader is invited to consult a formal and comprehensive description of the ASTD notation in [FGLFdf].

4. Refinement

It is a good practice to design the general behaviour of an IS at an abstract level before detailing the model. Often, the abstract model is composed of fewer states and transitions than the concrete model. The refinement process details states and transitions by introducing new transitions and new states. In this section, we describe our notion of refinement using a very abstract viewpoint, by considering only the traces of a system. Since it is defined on traces, it can be used for any notation which has a trace semantics.

4.1. Preliminary definitions

Let $\alpha(A)$ denote the set of automaton arrow labels occurring in ASTD A. Let a walk $u = q_0 \xrightarrow{\sigma_1} q_1 \xrightarrow{\sigma_2} \ldots \xrightarrow{\sigma_n} q_n$ for a trace $s = [\sigma_1, \sigma_2, \ldots, \sigma_n]$ denote the acceptance of s by an ASTD A, *i.e.*, q_0 is the initial state of ASTD A, and each transition $q_i \xrightarrow{\sigma_{i+1}} q_{i+1}$ is valid in A. The extension of the transition relation to traces is noted $q_0 \xrightarrow{s}_* q_n$, with $q_0 = q_n$ when s is the empty trace (*i.e.*, $s = []$). Let $A \setminus \Sigma$ be the hiding operator of CSP, which renames transitions over elements of Σ into the silent action τ (*i.e.*, $q_i \xrightarrow{\sigma} q_{i+1}$ in A becomes $q_i \xrightarrow{\tau} q_{i+1}$ in $A \setminus \Sigma$ when $\sigma \in \Sigma$). Action τ cannot be used to label automaton transitions in an ASTD; thus, we have that $\tau \notin \alpha(A)$ always holds for any ASTD A. Let $\mathsf{traces}(A)$ denotes the traces over $\alpha(A)$ of ASTD A; transitions on τ actions, which may be introduced by the hiding operator, are ignored when computing traces of an ASTD. As in CSP, let $\mathsf{divergences}(A)$ be the set of traces s such that A can execute an infinite sequence of τ transitions after executing a walk of trace s. Let $\mathsf{deadlocks}(A)$ be the set of traces s such that A cannot execute any action after executing a walk of trace s. Finally, let $\tau\text{-}\mathsf{enabled}(A)$ denote the set of traces s where A can execute a τ transition after executing a walk of s. Finally, let $A^{\setminus\overline{C}} = A \setminus (\alpha(A) - \alpha(C))$.

4.2. Refinement definitions

Definition 1 We say that $A \sqsubseteq^{\circlearrowleft} C$ iff

$$\alpha(C) - \alpha(A) \neq \varnothing \tag{1}$$

$$\mathsf{traces}(A^{\setminus\overline{C}}) = \mathsf{traces}(C^{\setminus\overline{A}}) \tag{2}$$

$$\mathsf{deadlocks}(A^{\setminus\overline{C}}) = \mathsf{deadlocks}(C^{\setminus\overline{A}}) \tag{3}$$

$$\tau\text{-}\mathsf{enabled}(A^{\setminus\overline{C}}) \subseteq \tau\text{-}\mathsf{enabled}(C^{\setminus\overline{A}}) \neq \varnothing \ . \tag{4}$$

We say that $A \sqsubseteq C$ iff $A \sqsubseteq^{\circlearrowleft} C$ and

$$\mathsf{divergences}(A^{\setminus\overline{C}}) = \mathsf{divergences}(C^{\setminus\overline{A}}) \, . \tag{5}$$

□

An ASTD A is refined by ASTD C when C introduces new actions wrt A; these new actions, called *refining actions* and represented by $\alpha(C) - \alpha(A)$, are used to refine the actions of A which do not occur in C, *i.e.*, actions represented by $\alpha(A) - \alpha(C)$ and called the *abstract actions*. Note that $\alpha(A) \subset \alpha(C)$ is sufficient for refinement; this occurs when there are no abstract actions to refine, and only refining actions are introduced. Note that we do require the introduction of refining actions by condition (1).

To compare two ASTDs with different alphabets, an easy solution is to hide abstract and refining actions, compare resulting traces, and check that abstract and refining actions behave in similar ways by looking at deadlocks and enabled actions. Expression $A^{\setminus\overline{C}}$ hides in A the events which are new with respect to C (*i.e.*, abstract actions which are replaced by refining actions). When refinement only introduces refining actions, *i.e.*, when $\alpha(A) \subset \alpha(C)$, nothing is hidden ($A^{\setminus\overline{C}} = A$). In other words, $A^{\setminus\overline{C}}$ is the restriction of A to $\alpha(A) \cap \alpha(C)$. Thus, by (2), we require A and C to have the same set of traces when abstract and refining actions are hidden. Condition (3) ensures that the refining actions do not introduce new deadlocks. Condition (4) ensures that when a sequence of abstract actions is executable in A, then a sequence of refining actions is also executable in C. In divergence-free refinement $\sqsubseteq$, C can loop forever on refining actions iff A can loop forever on some abstract actions. This ensures that terminating scenarios of the abstract system also terminate in the concrete system. Note that we do not use refusals to characterise our notion of refinement, because one abstract action is refined by several refining actions. Refusals are typically used to capture refinement by reducing nondeterminacy; we simply ensure that the concrete ASTD does not introduce new deadlocks, and that if an abstract action can be executed, then refining actions can also be executed, using the τ-enabledness condition (4).

To illustrate the notion of refinement, consider the following regular expressions which concisely represent the traces of some simple ASTDs.

$$A_1 = a, \quad A_2 = a \cdot c, \quad A_3 = a \cdot (b \mid c), \quad A_4 = a \cdot b^* \cdot c, \quad A_5 = a \cdot c \cdot d^*$$

The following refinements hold:

$$A_1 \sqsubseteq A_2,\quad A_1 \sqsubseteq A_3,\quad A_1 \sqsubseteq^{\circlearrowleft} A_4,\quad A_1 \sqsubseteq^{\circlearrowleft} A_5,$$
$$A_2 \sqsubseteq^{\circlearrowleft} A_4,\quad A_2 \sqsubseteq^{\circlearrowleft} A_5$$

A_2 is not refined by A_3, because of the following facts:

$$\begin{aligned}\mathsf{deadlocks}(A_2^{\setminus\overline{A_3}}) &= \{[a, c]\}\\ \mathsf{deadlocks}(A_3^{\setminus\overline{A_2}}) &= \{[a], [a, c]\}\end{aligned}$$

$A_3^{\setminus\overline{A_2}}$ can deadlock after executing $[a]$ due to the trace $[a, b]$ of A_3, which after hiding b becomes the trace $[a]$ in $A_3^{\setminus\overline{A_2}}$, while $A_2^{\setminus\overline{A_3}}$ does not deadlock after $[a]$. A_4 is not refined by A_5 because of the following facts:

$$\begin{aligned}\tau\text{-}\mathsf{enabled}(A_4^{\setminus\overline{A_5}}) &= \{[a]\}\\ \tau\text{-}\mathsf{enabled}(A_5^{\setminus\overline{A_4}}) &= \{[a, c]\}\end{aligned}$$

$A_4^{\setminus\overline{A_5}}$ can execute τ after the trace $[a]$, but $A_5^{\setminus\overline{A_4}}$ can't. Similarly, $A_5^{\setminus\overline{A_4}}$ can execute τ after the trace $[a, c]$, but $A_4^{\setminus\overline{A_5}}$ can't. Note that A_5 can be derived from A_4 by removing action b and inserting action d; it does not constitute a valid refinement because b is not replaced by d in the traces. Finally, $A_2 \not\sqsubseteq A_4$ and $A_2 \not\sqsubseteq A_5$ hold because of the following facts:

$$\begin{aligned}\mathsf{divergences}(A_2^{\setminus\overline{A_4}}) &= \mathsf{divergences}(A_2^{\setminus\overline{A_5}}) = \varnothing\\ \mathsf{divergences}(A_4^{\setminus\overline{A_2}}) &= \{[a]\}\\ \mathsf{divergences}(A_5^{\setminus\overline{A_2}}) &= \{[a, c]\}\end{aligned}$$

Our definitions of refinement use equality between sets of traces, instead of set inclusion like in refinement relations of well-known notations like CSP and EVENT-B. One may be tempted to call it an equivalence relation; it is very close to an equivalence relation, but it is not, due to condition (1), and not antisymmetric when $\alpha(A) - \alpha(C) \neq \varnothing$ and $\alpha(C) - \alpha(A) \neq \varnothing$. Thus, it is neither a partial order nor a pre-order. In plain language, refinement implies transforming a coarse object into something finer. This is what our refinement relation captures: some actions of A are refined into new actions in C, or simply new actions are introduced in C. For the sake of simplicity, C may have fewer refining actions than A has abstract actions. This is not especially important, since we are mainly interested in the preservation of behaviour: when some abstract action is executed in the abstract system, some refining actions are executed in the concrete system, and vice-versa. This concern arises from the actual practice in requirements analysis of IS. Scenarios are described in a stepwise fashion: some actions are abstracted into a high-level action t, which will be later refined, or abstracting low-level actions that may occur when some state has been reached. Traces of the abstract system are never "lost" during refinement: Abstract actions are simply replaced by concrete actions. This is why we use equality instead of set inclusion in our definitions.

5. Refinement patterns

Our refinement patterns arise from IS case studies, and they address typical characteristics of IS. Entities and associations play an important role in IS requirements analysis. Producing a data model describing entities and their associations is often the starting point of an IS requirements analysis process. The behaviour of an entity is typically defined using a "producer-modifier-consumer" life cycle [FSD03], as illustrated by automaton PMC-1 in Fig. 2. A producer action creates an instance of an entity; modifier actions allow this instance to evolve; a consumer action terminates the life of an instance. In the design process, producers, modifiers and consumers are often abstracted by using a single abstract action for each. For instance, it takes a complex workflow of actions to manage a flight ticket. In the initial stages of the refinement process, three simple actions like create-ticket, modify-ticket and terminate-ticket can be used to abstractly represent the life cycle of a flight ticket. Then, complex scenarios can be iteratively analyzed by refining these actions. For instance, a ticket can be terminated through alternative scenarios (the ticket is used, the flight is cancelled, the ticket is exchanged for another one, etc).

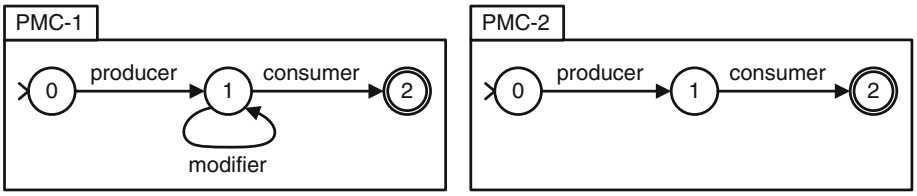

Fig. 2. Producer-Modifier-Consumer life cycle for entities and associations

Associations between entities are refined using a similar strategy. For instance, a book loan in a library system is an association between a member and a book. It is created by a lend action, modified by a renew action, and terminated by a return action. In such a refinement process, the following three refinement patterns frequently occur.

1. State refinement: In a black-box approach, the complexity of a state is ignored, and later detailed at the concrete level. New concrete actions are introduced to describe the inner workings of an abstract state.
2. Transition refinement: At the abstract level, a complex flow of actions is represented by using a single abstract transition. Refinement allows for the replacement of this abstract transition by new concrete actions.
3. Loop-transition refinement: This pattern is typically used to refine the modifier of an entity, for instance the modifier of automaton PMC-1 in Fig. 2. The effect of the loop-transition pattern could also be obtained by successive application of state refinement and transition refinement. For instance, state 1 of automaton PMC-2 in Fig. 2 could be refined to introduce an inner transition on the modifier, and then this modifier transition can be further refined by transition refinement. However, this has the disadvantage of not exhibiting right from the first model the existence of a modifier; the absence of the modifier restricts the scenarios that can be played with the users when simulating the system for initial validation. Thus, the loop-transition pattern has its own merit for refinement and validation.

These three patterns are subject to the following constraint: actions introduced by refinement in a concrete ASTD C must not exist in the abstract ASTD A that C refines. This can be formalized as follows. Let T_c be the set of transitions introduced in C using one of the three refinement patterns. The following two constraints must hold:

PC1. $T_c \neq \varnothing$

PC2. $\alpha(T_c) \cap \alpha(A) = \varnothing$

These patterns are defined for pure ASTDs as defined in [FGL+08, FGLFdf] (*i.e.*, ASTDs without state variables that can be modified by actions). These patterns can be extended to ASTDs with state variables. In the sequel, we shall briefly hint at the impacts of adding state variables.

The three refinement patterns are illustrated with the design of a system managing complaints of customers about their orders. A customer can complain as long as he has ordered something and he is still registered. Following the natural refinement process for this case study, the first pattern used is is loop-transition refinement, followed by state refinement and then transition refinement.

5.1. Loop-transition refinement

The loop-transition refinement pattern is presented in Fig. 3. Let A and C respectively be the abstract and concrete automaton ASTDs. Let s_a of type Elem be a state of A and t be a loop transition on s_a. Let T_a be the transitions of A. C is obtained from A by removing transition t and replacing state s_a with a state s_c of type Closure. Let J be the set of indices such that $\{out_j \mid j \in J\}$ is the set of transitions leaving s_c, but excluding t. When conditions LT1 to LT4 described below hold, we say that $A \sqsubseteq C$.

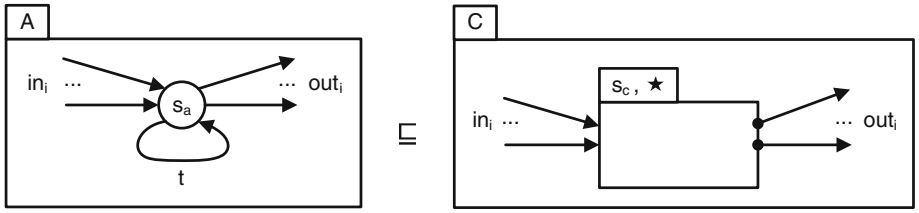

Fig. 3. Loop-transition pattern

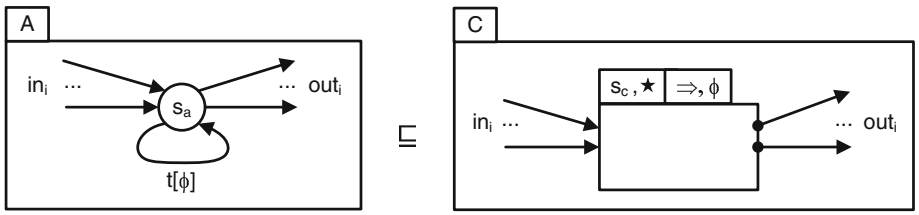

Fig. 4. Loop-transition pattern when transition t is guarded

LT1. Every transition leaving s_c must be final.

$$\forall j \in J : out_j.final? = \textbf{true}$$

This condition states that an out_j transition can be triggered only when ASTD s_c has reached a final state. This ensure that if the body of s_c is started, then it must terminate (*i.e.*, reach a final state) before executing an out_j transition.

LT2. In all states of all paths starting from the initial state q_0 of s_c, there is a path leading to a final state q of s_c.

$$q_0.L(s_c)^* = M(s_c) \ \wedge \ \forall s \in M(s_c) \ : \ \exists q \in F(s_c) \ : \ q \in s.L(s_c)^*$$

This conditions states that the body of s_c can terminate and an out_j can be triggered. The first conjunct ensures that all states of s_c are reachable from its initial state. The second conjunct ensures that any state of s_c can lead to a final state.

LT3. If there is a guard ϕ on the transition t, then the body of the closure in s_c must be of type Guard with condition ϕ, as illustrated in Fig. 4.

This conditions ensures that the first action of s_c can be triggered only when ϕ hold, to replicate the constraint in s_a that t can be triggered only when ϕ holds. The first action of a guard ASTD can be executed iff the guard holds and its inner ASTD can execute the action.

LT4. The action $\alpha(t)$ must not appear on any other transition of A.

$$\alpha(t) \cap \alpha(T_a - \{t\}) = \varnothing$$

This conditions ensures that all occurrences of $\alpha(t)$ are replaced.

Note that since $A^{\backslash \overline{C}}$ can diverge because of t, loop-transition refinement does not make C diverge more than A. Therefore, divergence-free refinement is always obtained. This will be explained in the refinement proof of Sect. 6.1.

A variant of this pattern is to omit condition LT1. This allows an out_j transition with $out_j.final? = \textbf{false}$ to be triggered even if the body of s_c has not reached a final state. For example, this can be used for "cancel" actions, whose purpose is to terminate an instance of an entity irrespective of the current state of s_c. Another variant is to relax condition LT4 by allowing $\alpha(t)$ to occur on other transitions of A and by replacing all of them. In that case, special care must be taken depending on whether $\alpha(t)$ occurs on non-loop transitions.

When ASTDs are extended with state variables that can be modified by actions, then guards of out_j transitions may refer to them. In that case, if the refining transitions can modify these state variables, then it must be ensured that when an out_j guard holds in s_a, then it also holds when s_c is in a final state, and vice-versa. This always hold in ASTD without state variables (*i.e.*, pure ASTDs), because guards of out_j transitions can only refer to variables of A and C; they cannot refer to local variables of s_c.

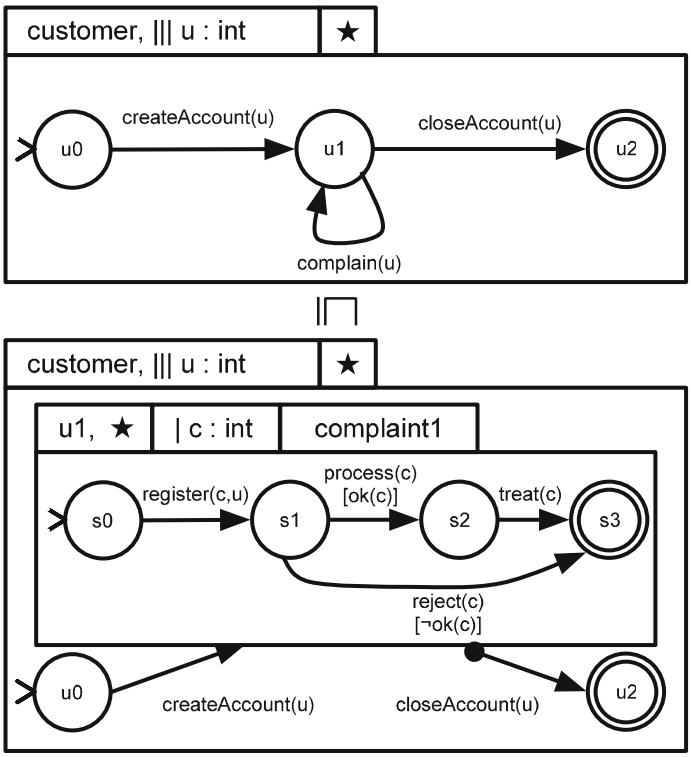

Fig. 5. Abstract model of the complaint IS and its first refinement using the *loop-transition* pattern

Case study—abstract model: complaining customers In Fig. 5, we present from top to bottom the first model of the IS and its first refinement. The first model is the most abstract of our refinement process. It describes how the customers are modelled in the system. The process is described using a quantified interleave (||| u) ASTD that models all *customer* entities of the system. A quantified interleave $||| \ x : \{v_1, \ldots, v_n\} : A$ interleaves n copies of ASTD A; it denotes $A[x := v_1] \ ||| \ \ldots \ ||| \ A[x := v_n]$. Each customer is associated to a value of u, an identifier used in actions of the ASTD. In order for complaints to be issued against the company, first the customer must be registered using the action createAccount(u). Then, the customer is in a state such that the system can accept one complaint complain(u) at a time. Finally, after the sequential resolution of zero, one or more complaints, the customer's account can be closed by the action closeAccount(u).

We now refine the elementary state u1 and the transition complain(u) using the loop-transition refinement pattern. The complaint process consists in registering the complaint c associated to customer u. Variable c is declared using a quantified choice ASTD, which makes the body of the closure; this variables is visible in the automaton making the body of the quantified choice. Either the complaint is accepted ([ok(c)]) and the complaint is treated by executing action treat(c), or the complaint is rejected. Conditions LT1 and LT3 are easily checked for this example. Condition LT2 is satisfied because the disjunction of the guards of transitions process(c) [ok(c)] and process(c) [¬ok(c)] always hold, hence there is always a path that leads to final state s3.

5.2. State refinement

The automaton state refinement pattern is presented in Fig. 6. It shares several conditions with the loop-transition pattern. Let A and C respectively be the abstract and concrete automaton ASTDs. Let s_a of type Elem be a state of A. C is obtained from A by replacing s_a with a state s_c of any ASTD subtype, except Elem. When conditions SR1 to SR2 described below hold, we say that $A \sqsubseteq^{\circ} C$; if condition SR3 also holds, then $A \sqsubseteq C$.

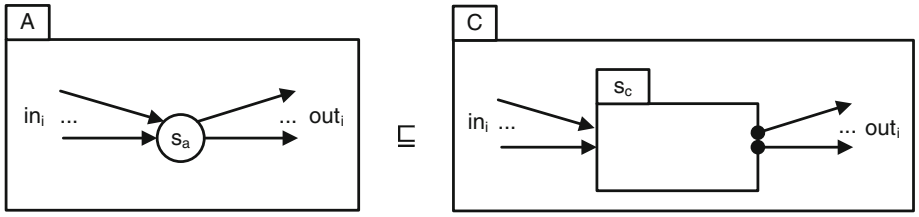

Fig. 6. Automaton state refinement pattern

SR1. Every transition leaving s_c must be final.

$$\forall j \in J : out_j.final? = \textbf{true}$$

SR2. In all states of all paths starting from the initial state q_0 of s_c, there is a path leading to a final state q of s_c.

$$q_0.L(s_c)^* = M(s_c) \ \wedge \ \forall s \in M(s_c) \ : \ \exists q \in F(s_c) \ : \ q \in s.L(s_c)^*$$

SR3. ASTD s_c contains no cycle.

$$I(M(s_c)) \cap L(s_c)^+ = \varnothing$$

Conditions SR1 and SR2 are exactly the same as conditions LT1 and LT2 of the loop-transition pattern, since the latter includes a form of state refinement. Condition SR3 is necessary if the specifier wants to exclude infinite loops on new actions, which is captured by divergence-free refinement. In state refinement, the refining state can be of any ASTD subtype, whereas the loop-transition pattern specifically requires refining with a Closure. Moreover, the existence of a loop transition on s_a is not required. If there is a loop transition, it is not refined, contrary to the loop-transition pattern.

Case study: evaluating complaint and taking into account feedback from the customer In Fig. 7, we present from top to bottom the ASTD complaint1 of the previous step shown in Fig. 5 and its refinement following the automaton state pattern. For the sake of concision, we do not reproduce all the ASTDs presented at the previous step. We only refine the complaint1 ASTD. The automaton state s1 is refined into an interleave of two AutomatonASTD that model two tasks: the first one is to take into account feedback from the customer in order to treat the complaint. The second one is to evaluate the relevance of the complaint. Once both tasks are completed, whatever the order, the complaint is either processed or rejected, as it was in the complaint1 ASTD. Condition SR1 is easily checked. Condition SR2 also hold: the interleave can reach a final state since each automaton of the interleave includes a final state reachable from all states. Since the refinement contains no cycle, condition SR3 hold and we have $\mathsf{complaint1}_a \sqsubseteq \mathsf{complaint1}_c$.

5.3. Transition refinement

The transition refinement pattern is presented in Fig. 8. Let A and C respectively be the abstract and concrete automaton ASTDs. Let s_{out} and s_{in} be two states of A connected by a transition t of type loc. Let T_a be the transitions of A. C is obtained from A by replacing transition t with a graph G of new transitions and new states wrt A, except for s_{out} and s_{in} which should appear in G. These new states are of type Elem. When conditions TR1 to TR3 described below hold, we say that $A \sqsubseteq^{\circ} C$; if condition TR4 also holds, then $A \sqsubseteq C$.

TR1. The action $\alpha(t)$ must not appear on any other transition of A.

$$\alpha(t) \cap \alpha(T_a - \{t\}) = \varnothing$$

TR2. If there is a guard ϕ on transition t, then ϕ must guard each new transition leaving s_{out} in G.

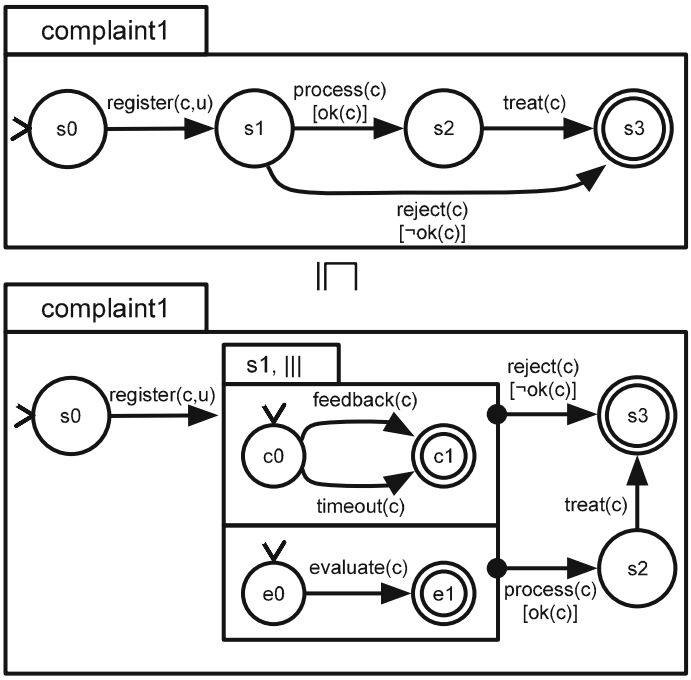

Fig. 7. Refinement of ASTD complaint1 using the *automaton state* pattern

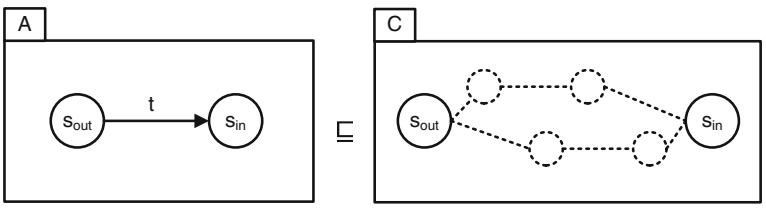

Fig. 8. Transition refinement pattern

TR3. s_{in} is reachable from any state of any path of G starting in s_{out}.

$$s_{out}.L(G)^* = M(G) \quad \wedge \quad \forall s \in M(G) \; : \; s_{in} \in s.L(G)^*$$

This condition is similar to condition LT2 of the loop-transition and state refinement patterns, with s_{out} playing the role of the initial state, and s_{in} playing the role of a final state.

TR4. The graph of new transitions contain no cycle.

$$I(M(G)) \cap L(G)^+ = \varnothing$$

This pattern could be extended to transitions of type tsub and fsub. The states introduced would have to be partitioned between the source automaton and the destination automaton of t.

The loop-transition refinement pattern is very similar, but not equivalent, to a combination of the state refinement pattern and the transition refinement pattern. The loop-transition pattern puts the refining transitions under the scope of a Closure (and possibly other embedded ASTD subtypes). A "producer-modifier-consumer" structure like the abstract ASTD of Fig. 5, cannot be refined into the same concrete ASTD using a combination of state refinement and transition refinement. State refinement could introduce a Closure, but it would not remove transition complain(u). Transition refinement would replace complain(u) by a graph of actions, but it would not allow the introduction of a QChoice to select a complaint c, which is a parameter of new action register(c,u), and it would not embed it into a compound state.

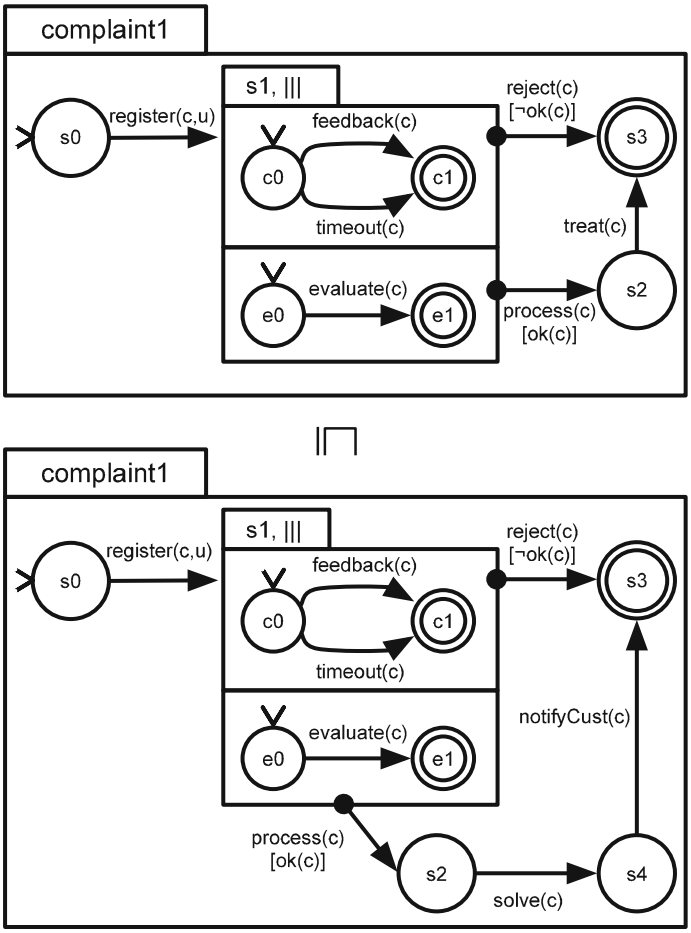

Fig. 9. Refinement of ASTD complaint1 using the *transition* pattern

Case study: solving issues and notifying the customer In Fig. 9, we present from top to bottom the ASTD complaint1 of the previous step shown in Fig. 7 and its refinement following the transition pattern. In this step, we would like to introduce two actions that will concretize the action treat(c). First we would like to solve the issue and then notify the customer that the issue has been corrected. The refinement of treat(c) introduces solve(c) and notify(c) that are executed in sequence. The state s4 is also introduced into this refinement and models the state when the issue has been solved but the customer has not been notified yet. These two actions are not present in the rest of the model and they will not take over the execution since the refinement does not introduce loops. We hence have $\mathsf{complaint1}_a \sqsubseteq \mathsf{complaint1}_c$.

6. Correctness of refinement patterns

We have proposed in Sect. 4 two definitions of refinement, in order to state what properties of a specification are preserved by refinement and how the abstract and concrete specifications are related. We now prove that each refinement pattern satisfies these definitions of refinement.

6.1. Loop transition refinement

All patterns rely on the syntactic replacement of one component e of ASTD A by a component e' to obtain ASTD C, *i.e.*, $C = A[e := e']$, such that the transitions labels introduced by e' are new; this establishes condition (1).

Condition (2) is established as follows. We first show $\mathsf{traces}(A^{\backslash\overline{C}}) \subseteq \mathsf{traces}(C^{\backslash\overline{A}})$. Let s be a trace of A. If s does not include t (the abstract transition), then C can also execute this transition, since the walk accepting s in A is still feasible in C; in the particular case where the abstract state p occurs in the walk, the semantics of a Kleene closure ASTD provides that its initial state is final, ensuring that the closure can be exited immediately through one of the out transition without any iteration in the concrete state. If t occurs in s, then the walk u accepting s can be used to construct a walk accepting $s \setminus \{t\}$ in C, by simply suppressing the steps $q_i \xrightarrow{t}$ in u and executing no loop in C over p. This shows that any trace of $A^{\backslash\overline{C}}$ is also a trace of $C^{\backslash\overline{A}}$. We now show $\mathsf{traces}(A^{\backslash\overline{C}}) \supseteq \mathsf{traces}(C^{\backslash\overline{A}})$. Let s be a trace of C. The case where the trace does not contain any refining action is similar to the case above where t does not occur in s. When s contains refining actions, then s can be decomposed as follows: $s = w_0 \cdot v_1 \cdot w_1 \cdot \ldots v_n \cdot w_n$ (where $\cdot$ denotes trace concatenation), such that $v_i \in (\alpha(A) \cap \alpha(C))^+$, and $w_i \in (\alpha(C) - \alpha(A))^+$ for $i \in 1..n-1$ and $w_0, w_n \in (\alpha(C) - \alpha(A))^*$. Thus, the walk

$$q_0 \xrightarrow{w_0}_* q_1 \xrightarrow{v_1}_* q_2 \xrightarrow{w_1}_* \ \ldots\ q_{k-2} \xrightarrow{v_n}_* q_{k-1} \xrightarrow{w_n}_* q_k$$

in C can be used to construct the walk

$$\widehat{q_0} \xrightarrow{t} \widehat{q_1} \xrightarrow{v_1}_* \widehat{q_2} \xrightarrow{t} \ \ldots\ \widehat{q_{k-2}} \xrightarrow{v_n}_* \widehat{q_{k-1}} \xrightarrow{t} \widehat{q_k}$$

in A, where $\widehat{q_i}$ denotes state q_i with s_c replaced by s_a, and with $q_0 \xrightarrow{t}$ and $\xrightarrow{t} q_k$ omitted when $w_0, w_n = []$, respectively.

For deadlocks, let s be a deadlock of A; s does not end with t, since t is a loop. A corresponding trace exists in C (modulo the replacement of t by refining actions, as shown above). This trace also deadlocks in C, because the deadlocking state also exists in C, due to the construction of C from A by syntactic replacement. Thus, $\mathsf{deadlocks}(A^{\backslash\overline{C}}) \subseteq \mathsf{deadlocks}(C^{\backslash\overline{A}})$ holds. Now, consider a deadlock s of C. If s does not end with a refining action, then this case is similar to the one just considered for A. If s ends with a refining action with A not being in a final state of p, then this contradicts condition LT2 of the pattern: a final state is reachable from any state of s_c. If s ends with a refining action with C being in a final state of s_c, then there is no deadlock, since a new iteration on the closure s_c can be started. Thus $\mathsf{deadlocks}(C^{\backslash\overline{A}}) \subseteq \mathsf{deadlocks}(A^{\backslash\overline{C}})$ holds.

For divergences, the proof follows easily from the equality on traces. A divergence of $A^{\backslash\overline{C}}$ can be mapped to a divergence of $C^{\backslash\overline{A}}$, and vice-versa: it suffices to consider the walks of A which form a divergence in $A^{\backslash\overline{C}}$, map them to walks of C and a divergence of $C^{\backslash\overline{A}}$ is obtained; similarly for the opposite.

Condition (4) simply ensures that the abstract ASTD can execute an abstract action, then the concrete ASTD can execute concrete actions. It follows from pattern conditions LT2 and LT3, and from PC1.

6.2. State refinement and transition refinement

The proof of correctness for the state refinement pattern is very similar to the proof for loop transition refinement. The major difference is that there is no transition to replace, thus $\alpha(A) - \alpha(C) = \varnothing$. But it is quite easy to see that walks of A can be mapped to walks of C by inserting the refining transitions; similarly for traces of C that can be mapped to walks of A. Since C cannot deadlock in the refining state s_c by condition SR2, then deadlocks of A and C reside in their common parts unaffected by refinement. For divergences: $A^{\backslash\overline{C}}$ cannot diverge since there is nothing to hide in A; condition SR3 ensures that the refining state contains no loop, so $C^{\backslash\overline{A}}$ cannot diverge either. Finally, for τ-enabledness, pattern condition SR2 ensures that the refining state p can execute at least one transition; thus $\tau\text{-}\mathsf{enabled}(C^{\backslash\overline{A}}) \neq \varnothing$. Since $A^{\backslash\overline{C}}$ contains no hidden event, it has no enabled τ-transition; thus $\tau\text{-}\mathsf{enabled}(A^{\backslash\overline{C}}) = \varnothing$.

The proof of correctness for transition refinement is also very similar. Traces can be mapped from one to the other. By condition TR3, no new deadlocks are introduced in C wrt A. By trace correspondence, if $A^{\backslash\overline{C}}$ can diverge, than $C^{\backslash\overline{A}}$ can also diverge. If A has no cycle, then divergences of $A^{\backslash\overline{C}}$ are also divergences of $A^{\backslash\overline{C}}$ by trace

mapping. By condition TR3, C has at least one transition enabled and it is enabled iff A has abstract transition t enabled, because they have the same guards by condition TR2.

7. Congruence-like properties

Our refinement patterns can be applied to a sub-part of a specification. An important property that should be satisfied by refinement is that refining a part should refine the whole, which is typically called a congruence. Suppose that B is an ASTD which includes a sub-ASTD A. If A is refined by C, then does $B[A := C]$, which denotes the ASTD obtained by replacing A with C in B, refines B? It is not the case in general, since the new actions introduced in C may already exist in A; this could introduce new deadlocks if C occurs within a synchronization. However, when our refinement patterns are used under some conditions, then B is indeed refined by $B[A := C]$. The following theorems describe these refinements.

Theorem 7.1 Let B be an ASTD which includes a sub-ASTD A. Let C be an ASTD obtained from A by applying the loop-transition refinement pattern with satisfaction of its conditions. Let t the transition of A refined by applying the pattern. Let T_b be the transitions of B. Let T_c be the set of new transitions introduced in C. If the following conditions hold, then $B \sqsubseteq B[A := C]$.

1. $\alpha(B) \cap \alpha(T_c) = \varnothing$.
2. $\alpha(t) \cap \alpha(T_b - \{t\}) = \varnothing$.
3. For each $e \in \alpha(T_c \cup \{t\})$, e does not occur in any synchronization set Δ of a parent ASTD $(|[]|, n, \Delta, l, r)$ of A in B.

Proof The proof of this theorem is very similar to the proof of refinement for the loop-transition pattern in Sect. 6.1. Condition 1 is the extension to B of PC2 for A. Similarly, condition 2 is the extension of condition LT4 of the loop-transition pattern to B. The main difficulty lies with the occurrence of A as a child of an ASTD of type Synchronization, QSynchronization, or ASTDCall, which could introduce new deadlocks. If A occurs within a synchronization of the form $\langle |[]|, n, \Delta, l, r\rangle$, condition 3 ensures that no synchronization is required on actions of $\alpha(T_c \cup \{t\})$. If a parallel composition $\langle \|, n, l, r\rangle$, which is defined in the ASTD semantics as $\langle |[]|, n, \alpha(l) \cap \alpha(r), l, r\rangle$, occurs as a parent of A, then no synchronization is required on $\alpha(T_c \cup \{t\})$ in either B or $B[A := C]$, because of the following: by 2, no synchronization is required on $\alpha(t)$ in B; by 1, no synchronization on $\alpha(T_c)$ in $B[A := C]$. The last case to consider is a call $\langle \mathsf{cal}, n, P(\vec{v})\rangle$ where P is a definition $P(\vec{x} : \vec{T}) \triangleq A$. If this call occurs within a parallel composition $\langle \|, n, l, r\rangle$, then if l and r were synchronizing on t in B, then they can also synchronize on the sequences of actions replacing t in $B[C := A]$. □

Similar theorems hold for the state refinement pattern and the transition refinement pattern. Their proof is omitted.

Theorem 7.2 Let B be an ASTD which includes a sub-ASTD A. Let C be an ASTD obtained from A by applying the transition refinement pattern. Let t the transition of A refined by applying the pattern. Let T_b be the transitions of B. Let T_c be the set of new transitions introduced in C. If the following conditions hold:

1. $\alpha(B) \cap \alpha(T_c) = \varnothing$,
2. $\alpha(t) \cap \alpha(T_b - \{t\}) = \varnothing$, and
3. for each $e \in \alpha(T_c \cup \{t\})$, e does not occur in any synchronization set Δ of a parent ASTD $(|[]|, n, \Delta, l, r)$ of A in B,

then

$$A \sqsubseteq^{\circ} C \quad \Rightarrow \quad B \sqsubseteq^{\circ} B[A := C]$$

and

$$A \sqsubseteq C \quad \Rightarrow \quad B \sqsubseteq B[A := C] \ .$$

Theorem 7.3 Let B be an ASTD which includes a sub-ASTD A. Let C be an ASTD obtained from A by applying the state refinement pattern. Let T_c be the set of new transitions introduced in C. If $\alpha(B) \cap \alpha(T_c) = \varnothing$, then

$$A \sqsubseteq^{\circlearrowleft} C \quad \Rightarrow \quad B \sqsubseteq^{\circlearrowleft} B[A := C]$$

and

$$A \sqsubseteq C \quad \Rightarrow \quad B \sqsubseteq B[A := C] \ .$$

8. A comparison with refinement in EVENT-B and CSP

As stated in the introduction, a large number of refinement relations have been proposed. We are interested in refinement for constructing a system specification. We have chosen CSP and EVENT-B for a comparison, because these two methods are well known representative of refinement methods for specification construction. CIRCUS and CSP ‖ B use the same refinement relation as CSP (they also use total correctness refinement and data refinement, which we do not use). For a comparison between CSP refinement and EVENT-B refinement, see [STW11]. ASTD specification can be translated into EVENT-B specifications [MFGL10] and B specifications for implementation [Mil11a].

8.1. A comparison with CSP

The three main refinement relations used in CSP are trace refinement ($\sqsubseteq_T$), failure refinement ($\sqsubseteq_F$) and failure-divergence refinement ($\sqsubseteq_{FD}$) [RHB98]. A process expression $A \sqsubseteq_T C$ iff $\mathsf{traces}(C) \subseteq \mathsf{traces}(A)$; thus it allows loss of traces during refinement, contrary to our definition. Moreover, trace refinement does not allow to add new actions in refinement, contrary to our definition. Failure refinement considers *failures*, *i.e.*, pairs (s, Σ) such that a process can refuse all actions of Σ after executing trace s and reaching a *stable state*, that is, a state with no outgoing τ transition. A process expression $A \sqsubseteq_F C$ iff $\mathsf{failures}(C) \subseteq \mathsf{failures}(A)$. Thus, failures are useful for us to detect deadlocks that are introduced in the concrete ASTD C: a deadlock is a failure $(s, \alpha(C))$. Failure-divergence refinement considers divergences mixed with failures: $A \sqsubseteq_{FD} C$ iff $\mathsf{failures}_\perp(C) \subseteq \mathsf{failures}_\perp(A)$ and $\mathsf{divergences}(C) \subseteq \mathsf{divergences}(A)$, where $\mathsf{failures}_\perp(A) = \mathsf{failures}(A) \cup \{(s, X) \mid s \in \mathsf{divergences}(A)\}$. Divergences are useful to capture loops introduced in the refining process expression. When $A \sqsubseteq_R A' \wedge A' \sqsubseteq_R A$, we write $A =_R A'$ with $R \in \{T, F, FD\}$.

The first question that comes to mind is whether there exists a simple connection between ASTD refinement and the traditional CSP refinements. The next two entailments (6) and (7) illustrate this very close connection.

$$A \sqsubseteq^{\circlearrowleft} C \quad \Rightarrow \quad A^{\setminus \overline{C}} =_F C^{\setminus \overline{A}} \tag{6}$$

$$A \sqsubseteq C \quad \Rightarrow \quad A^{\setminus \overline{C}} =_{FD} C^{\setminus \overline{A}} \tag{7}$$

For instance in Fig. 10, where the transition refinement pattern is applied, we have that $A1 \sqsubseteq C1$ and $A1^{\setminus \overline{C1}} =_{FD} C1^{\setminus \overline{A1}}$. When the shaded transition $5 \xrightarrow{\mathsf{b1}} 6$ of $C2$ is not taken into account (for the sake of illustration), we have that $A1 \sqsubseteq^{\circlearrowleft} C2$, $A1 \not\sqsubseteq C2$, $A1^{\setminus \overline{C2}} =_F C2^{\setminus \overline{A1}}$ and $A1^{\setminus \overline{C2}} \neq_{FD} C2^{\setminus \overline{A1}}$, since $C2$ contains a cycle on $\mathsf{b3}$ which introduces a divergence after hiding $\mathsf{b3}$. If the shaded transition is taken into account, then both $A1 \not\sqsubseteq^{\circlearrowleft} C2$ and $A1^{\setminus \overline{C2}} \neq_F C2^{\setminus \overline{A1}}$ hold, since this shaded transition introduces a deadlock in $C2^{\setminus \overline{A1}}$ which is captured by $A1^{\setminus \overline{C2}} \not\sqsubseteq_F C2^{\setminus \overline{A1}}$.

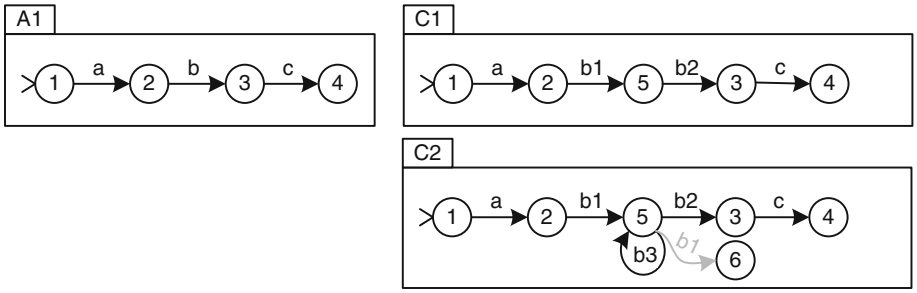

Fig. 10. Examples where $=_F$ and $=_{FD}$ modulo hiding coincide with $\sqsubseteq^{\circlearrowleft}$ and $\sqsubseteq$, respectively

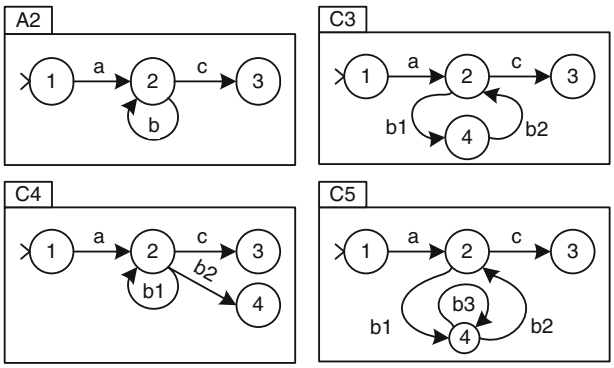

Fig. 11. Examples where $=_F$ and $=_{FD}$ modulo hiding do not coincide with $\sqsubseteq^{\circlearrowleft}$ and $\sqsubseteq$

The examples of Fig. 10 show where ASTD refinement and CSP equivalence modulo hiding coincide; Fig. 11 shows some peculiar cases where they do not coincide. We have that $A2 \sqsubseteq C3$ and $A2^{\setminus \overline{C3}} =_{FD} C3^{\setminus \overline{A2}}$, and similarly for $A2$ and $C5$, but the equivalence between $\sqsubseteq$ and $=_{FD}$ fails for $A2$ and $C4$: we have that $A2 \not\sqsubseteq^{\circlearrowleft} C4$, but $A2^{\setminus \overline{C4}} =_{FD} C4^{\setminus \overline{A2}}$. The discrepancy in $C4$ arises from diverging state 2, which adds failure $([a], \alpha(A) \cap \alpha(C))$ to $\mathsf{failures}_\perp$ for all ASTDs of Fig. 11 when hiding the abstract and refining actions. Thus, the deadlock introduced by transition b2 in $C4$ is not detected as a new failure wrt $A2$. In summary, failure refinement and failure-divergence refinements are very close to ASTD refinement, but they do not discriminate enough for some peculiar cases like when $s_{out} = s_{in}$ in the transition refinement pattern. Moreover, they do not take into account τ-enabledness; σ-enabledness is usually checked with failure refinement in CSP (*i.e.*, checking that σ never occurs in a failure), but it does not apply for $\sigma = \tau$. When state refinement is used, CSP equivalence (modulo hiding) and ASTD refinement do coincide.

Note also that our definition of refinement is slightly weaker than the conditions of the transition refinement pattern. We have that $A2 \sqsubseteq C5$, but $C5$ does not satisfy condition TR4 of the transition refinement pattern, since it contains a loop within the refining transitions. But since $A2$ also contains a loop, then the divergences induced by the refining transitions are not new divergences wrt $A2^{\setminus \overline{C5}}$.

8.2. EVENT-B

EVENT-B [Abr10] is a state-based, event-driven modelling notation. EVENT-B models, called machines, are developed through stepwise refinement. A machine has a *state* defined by state variables, *invariants*, which describe invariant properties of states, and *events*, which can be triggered when their guard is satisfied and whose execution can modify the machine state. One must prove that each event execution preserves the invariants. EVENT-B refinement allows for *behaviour refinement* (*i.e.*, reducing non-determinism, guard strengthening, event splitting/merging and introduction of new events) and for *data refinement* (*i.e.*, adding new state variables and replacing state variables). To prove that a machine M_1 is refined by a machine M_2, noted here $M_1 \sqsubseteq_B M_2$, one must prove the following.

MACHINE A1

VARIABLES s

INVARIANTS $s \in 1 \,..\, 4$

EVENTS
EVENT INITIALISATION THEN $s := 1$ END

EVENT a WHERE $s = 1$ THEN $s := 2$ END

EVENT b WHERE $s = 2$ THEN $s := 3$ END

EVENT c WHERE $s = 3$ THEN $s := 4$ END
END

Fig. 12. Machine $A1$ which corresponds to ASTD $A1$ of Fig. 10

1. Guard strengthening: for each event e_1 refined by event e_1', show that the guard of e_1' entails the guard of e_1; when an event is refined by several events (*i.e.*, event splitting), show that the disjunction of the guards of the refining events entails the guard of the abstract event.
2. Simulation: each execution of a refining event must match an execution of its abstract event. Note that the opposite does not need to hold: an abstract event execution does necessarily need to be matched by an execution of its refining events. This is usually called a downward simulation. When data refinement is used, the concrete machine must contain a gluing invariant that relates the state of the abstract machine to the state of the concrete machine; this relationship need not be a function. New events are considered to refine an implicit skip action which preserves the value of the state of the abstract machine; thus, new events can only modify new state variables while satisfying the gluing invariant.
3. Convergence: show that the concrete machine cannot execute an infinite sequence of new events. This is shown by exhibiting a variant (a natural number or a finite set) and proving that this variant is decreased by each execution of new events.

EVENT-B is supported by a set of tools, including the Rodin platform, which automatically generates proof obligations for proving refinement. In [Abr10], other proof obligations are described, and they can be manually included as theorems to be proved from the invariants of the machine. This includes proving relative deadlock freedom: the concrete machine should not introduce new deadlocks wrt to the abstract machine. This is proved by showing that the disjunction of the abstract guards entails the disjunction of the concrete guards.

Guard strengthening and downward simulation allow for the concrete machine to have less traces than the abstract machine, thus EVENT-B refinement does not entail ASTD refinement. On the other hand, ASTD refinement with no divergence ($\sqsubseteq$) still allows for infinite loops on new events when the abstract ASTD can loop on the abstract event; thus, ASTD refinement does not either entail B refinement. However, the three patterns do entail B refinement when cycles are excluded, provided that a proper gluing invariant, an appropriate event refinement mapping and a variant are devised.

We illustrate this EVENT-B refinement on ASTDs $A1$ and $C1$ of Fig. 10: Fig. 12 provides the straightforward translation of $A1$ to EVENT-B, while Fig. 13 represents its refinement $C1$. Abstract transition b is refined by two concrete events b1 and b2, which leads to the introduction of state 5. To simplify the derivation of the gluing invariant and the variant, we have chosen to represent this new state by adding a new variable u, which also stands as the variant. The transition system of the EVENT-B machine $C1$ is illustrated in Fig. 14: States are labelled by the pair s, u denoting the value of s and u. This way, guard strengthening, simulation and convergence become trivial to prove. In all patterns, with no cycles allowed, the transition relation on new states defines a strict order on new states, from which one can always derive a linear order on the new states, and use it to number the new states in decreasing order. A linear order can also be derived if a compound ASTD is used for refinement: orders of the inner ASTDs are combined to form a new linear order for the compound ASTD. The initial events, that is, those which are the first executable transition (*e.g.*, b1) are those selected to refine the abstract transition and declared as extended, which means that they inherit the guard and actions of the abstract event. This also simplifies the proof of simulation between abstract and concrete events. Subsequent transitions (*e.g.*, b2) are

MACHINE C1 REFINES A1

VARIABLES s u

INVARIANTS
$u \in 0\,..\,2$
$s = 1 \wedge u = 2 \;\vee\; s = 2 \wedge u = 2 \;\vee\; s = 3 \wedge u = 1 \;\vee\; s = 3 \wedge u = 0 \;\vee\; s = 4 \wedge u = 0$

VARIANT u

EVENTS
EVENT INITIALISATION EXTENDS INITIALISATION
THEN $u := 2$
END

EVENT a EXTENDS a END

EVENT b1 REFINES b
WHERE $s = 2 \wedge u = 2$ THEN $s := 3 \parallel u := 1$
END

CONVERGENT EVENT b2
WHERE $s = 3 \wedge u = 1$ THEN $u := 0$
END

EVENT c EXTENDS c
WHERE $u = 0$
END
END

Fig. 13. Machine $C1$ which refines (wrt $\sqsubseteq_B$) Machine $A1$ and corresponds to ASTD $C1$ of Fig. 10

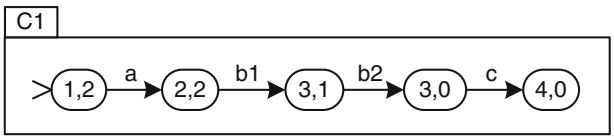

Fig. 14. The transition system of EVENT-B machine $C1$

declared as convergent new events. Finally, the transition following the refined transition (*e.g.*, c) is refined as an extension to take into account the new state variables in its guard. The relative deadlock freedom proof obligation is added as a theorem in the invariant; theorems are proved from invariants, whereas invariants are proved over all events. Relative deadlock freedom ensures that the final state or s_{in} is reachable from the initial state or s_{out} in the patterns.

9. Conclusion and future works

We have presented three refinement patterns for ASTDs. We have proposed an ASTD refinement relation which characterizes the effects of these refinement patterns on the behaviour of a system. These patterns allow designers to adopt a top-down approach for specifying IS, like the "parachute" paradigm in EVENT-B which consists in providing more and more details at each refinement step. We have taken advantage of a number of refinement concepts: (i) new events can be introduced as in EVENT-B refinement; (ii) single state and transition can be replaced by more complex ASTDs as in colored Petri-nets refinement; (iii) Statecharts refinement is extended by allowing for all ASTD types to be used in refinement, including the equivalents of OR- and AND-states. These patterns seem

to cover some of the most common cases encountered while specifying an IS, but they seem to general enough to be used in other application domains like telecommunication protocols and reactive systems. However they are neither exhaustive nor exclusive. Other kinds of patterns can be defined as, for example, the ones presented in [SK10, MNB04]. We intend to address them in the near future, to enrich our pattern library.

We have shown that our notion of refinement differs from refinements found in well established methods like CSP and EVENT-B. This stems from the chosen target application domain, information systems, and the practice of requirements definition in this domain. In this particular context, refinement means adding new details about events and states, but never losing scenarios of event execution. In CSP and EVENT-B, the view point is different: a specification is refined by adding new properties, which constrain system behaviour, thus some scenarios of the abstract level can be lost by adding new constraints, because they were invalid wrt to the new property added by refinement. Refinement in IS is mostly syntactic and structural: the system behaviour is described by adding new parts which are replacing "stubs" so to speak. Our refinement pattern conditions make sure that this insertion of new details does not contradict, reduce or transform the behaviour of the system; it only adds new sub-behaviours to it.

In this paper, we use pure ASTDs to specify the *ordering* of input events in an IS. ASTDs represent an alternative to UML diagrams which are often used to specify IS use cases and scenarios (*e.g.*, sequence diagram, activity diagram, collaboration diagram, and state machine diagram). We do not deal with *data management* requirements of an IS. ASTDs can be extended with state variables to deal with data management. Our patterns are concerned with *event refinement* in pure ASTDs. Other patterns must be defined to deal with *data refinement* if ASTDs extended with state variables are used. These data refinement patterns would be used in conjunction with our event refinement patterns. Our work addresses requirements specification; we do not deal with implementation issues like refining an action into database transactions or refining abstract data into database tables.

The semantics of our refinement is formal, but their proof obligations are not easy to carry out as defined. As a future work, we aim at providing a more systematic and practical method for discharging proof obligations for refinement. These could be carried out in EVENT-B and its toolset using the EVENT-B translation of ASTD [Mil11a] and by generating appropriate proof obligations, because, as we have shown, the EVENT-B standard proof obligations are not adequate for our definition of refinement. In addition, it seems that a purely syntactic and fully automatic verification could also be done for all operators except synchronization, which requires a proof when large quantifications are used. If ASTDs are augmented with state variables that could be modified in actions and tested in guards, then a proof approach *à la* EVENT-B will be necessary to fully discharge the proof obligations. Another extension of this work is the implementation of a graphical tool for IS modeling using ASTDs and taking the refinement patterns into account. This tool could include an animator and model checker in order to validate the model during the design phase.

Acknowledgements

This research is supported by ANR (France) as part of the SELKIS project (ANR-08-SEGI-018) and by NSERC (Canada). The authors would like to thank the anonymous referees for their comments, which lead to numerous improvements.

References

[Abr96] Abrial JR (1996) The B-book: assigning programs to meanings. Cambridge University Press, Cambridge
[Abr10] Abrial JR (2010) Modeling in Event-B. Cambridge University Press, Cambridge
[AtH12] Aalst WMP, ter Hofstede AHM (2012) Workflow patterns put into context. Softw Syst Model 11(3):319–323
[BGPS12] Bianculli D, Ghezzi C, Pautasso C, Senti P (2012) Specification patterns from research to industry: a case study in service-based applications. In: Proceedings of the 2012 international conference on software engineering. ICSE 2012, Piscataway, NJ, USA. IEEE Press, pp 968–976
[BKS83] Back RJR, Kurki-Suonio R (1983) Decentralization of process nets with centralized control. In: Proceedings of the 2nd ACM symposium on PODC, pp 131–142
[BKS88] Back R-J, Kurki-Suonio R (1988) Distributed cooperation with action systems. ACM Trans Program Lang Syst 10(4):513–554
[BvW94] Back RJR, von Wright J (1994) Trace refinement of action systems. In: Structured programming. Springer, Heidelberg, pp 367–384
[CMP08] Choppy C, Mayero M, Petrucci L (2008) Experimenting formal proofs of petri nets refinements. Electron Notes Theor Comput Sci 214:231–254
[Cop03] Coplien JO (2003) Software design patterns. In: Encyclopedia of computer science. Wiley, Chichester, pp 1604–1606

[DAC99] Dwyer MB, Avrunin GS, Corbett JC (1999) Patterns in property specifications for finite-state verification. In: Proceedings of the 21st international conference on Software engineering, ICSE '99, New York, NY, USA. ACM, pp 411–420

[DvL96] Darimont R, van Lamsweerde A (1996) Formal refinement patterns for goal-driven requirements elaboration. In: Proceedings of the 4th ACM SIGSOFT symposium on foundations of software engineering, SIGSOFT '96, New York, NY, USA. ACM, pp 179–190

[EJFG+10] Embe Jiague M, Frappier M, Gervais F, Konopacki P, Milhau J, Laleau R, St-Denis R (2010) Model-driven engineering of functional security policies. In: Proceedings of the international conference on enterprise information systems 3:374–379

[FGL+08] Frappier M, Gervais F, Laleau R, Fraikin B, St-Denis R (2008) Extending statecharts with process algebra operators. Innov Syst Softw Eng 4(3):285–292

[FGLFdf] Frappier M, Gervais F, Laleau R, Fraikin B (2008) Algebraic state transition diagrams. Technical report 24, Département d'informatique, Université de Sherbrooke, Sherbrooke, QC, Canada http://www.dmi.usherb.ca/~frappier/Papers/astd2008.pdf.

[FSD03] Frappier M, St-Denis R (2003) Eb3: an entity-based black-box specification method for information systems. Softw Syst Model 2(2):134–149

[GHJV94] Gamma E, Helm R, Johnson R, Vlissides J (1994) Design patterns: elements of reusable Object-Oriented Software. 1st edn., Addison-Wesley Professional, Boston

[Gla90] van Glabbeek RJ (1996) Comparative Concurrency Semantics and Refinement of Actions. PhD thesis, Free University, Amsterdam, 1990. Second edition available as CWI tract 109, CWI, Amsterdam

[Har87] Harel D (1987) Statecharts: a visual formalism for complex systems. Sci Comput Programm 8(3):231–274

[MFGL10] Milhau J, Frappier M, Gervais F, Laleau R (2010) Systematic translation rules from ASTD to Event-B. In: Dominique M, Stephan M (eds) Integrated formal methods, vol 6396 of lecture notes in computer science. Springer, Berlin/Heidelberg, pp 245–259

[Mil11a] Milhau J (2011) Un processus formel d'intégration de politiques de contrôle d'accès dans les systèmes d'information. PhD thesis, Université de Sherbrooke–Université Paris-Est, Sherbrooke

[MIL+11b] Milhau J, Idani A, Laleau R, Labiadh M, Ledru Y, Frappier M (2011) Combining UML, ASTD and B for the formal specification of an access control filter. Innov Syst Softw Eng 7(4):303–313

[MNB04] Meng S, Naixiao Z, Barbosa LS (2004) On semantics and refinement of uml statecharts: a coalgebraic view. In: Proceedings of the 2nd international conference on software engineering and formal methods, SEFM '04, Washington, DC, USA. IEEE Computer Society, pp 164–173

[RHB98] Roscoe AW, Hoare CAR, Bird R (1998) The theory and practice of concurrency. Prentice Hall PTR, Upper Saddle River, NJ, USA

[Rut00] Rutten J (2000) Universal coalgebra: a theory of systems. Theoret Comput Sci 249:3–80

[Sai56] Said MY (2010) Methodology of refinement and decomposition in UML-B. PhD thesis, University of Southampton, Southampton. http://eprints.ecs.soton.ac.uk/21656/

[Sch98] Scholz P (1998) A refinement calculus for statecharts. In: Egidio A. (ed) Fundamental approaches to software engineering, vol 1382 of lecture notes in computer science. Springer, Berlin/Heidelberg, pp 285–301

[SK10] Schönborn J, Kyas M (2010) Refinement patterns for hierarchical uml state machines. In: Arbab F, Sirjani M (eds) Fundamentals of software engineering, vol 5961 of lecture notes in computer science. Springer, Berlin/Heidelberg, pp 371–386

[ST11] Schneider S, Treharne H (2011) Changing system interfaces consistently: a new refinement strategy for CSP ‖ B. Sci Comput Program 76(10):837–860

[STW11] Schneider S, Treharne H, Wehrheim H (2011) A csp account of Event-B refinement. In: Proceedings of the refinement workshop on EPTCS 55, pp 139–154

[WC02] Woodcock J, Cavalcanti A (2002) The semantics of circus. In: Bert D, Bowen JP, Henson MC, Robinson K (eds) ZB 2002: formal specification and development in Z and B, vol 2272 of lecture notes in computer science. Springer, Berlin/Heidelberg, pp 184–203

Received 2 October 2012

Revised 16 May 2013

Accepted 20 May 2013 by I. Perseil, P. Gibson, and J. Woodcock

Published online 22 August 2013