Science of Computer Programming 170 (2019) 45–79

Contents lists available at ScienceDirect

Science of Computer Programming

www.elsevier.com/locate/scico

Structure and behavior preserving statecharts refinements

Eugene Syriani [a,*], Vasco Sousa [a], Levi Lúcio [b]

[a] University of Montreal, P.O. Box 6128, succ. Centre-Ville, Montreal QC, H3C3J7, Canada
[b] fortiss GmbH, Guerickestraße 25, 80805 Munich, Germany

ARTICLE INFO

Article history:
Received 12 May 2017
Received in revised form 18 October 2018
Accepted 23 October 2018
Available online 6 November 2018

Keywords:
Statechart
Refinement
Software design
Verification

ABSTRACT

Statecharts are one of the most popular modeling formalisms that is used in a diversity of real world applications, such as cyber-physical systems, mobile computing, and bioinformatics. Following common step-wise refinement strategies, modelers often develop and evolve Statecharts models incrementally to satisfy requirements and changes. Although there are already several existing Statecharts refinement approaches, they provide insufficient guarantees to support reasoning about the preservation of both its internal structure, to enhance development and maintenance, and its behavior, which external components using the Statecharts model may depend on. In this paper, we formulate formal requirements to minimally limit the allowable Statecharts refinement operations such that certain assumptions on the structure and reachability of the Statecharts model will hold. To satisfy these requirements, we propose a set of practical bounding refinement rules that allows the modeler to achieve such refinements for which notions of behavior and structure preservation can be statically verified under some assumptions. Our implementation shows how these rules can be applied in realistic scenarios.

1. Introduction

The Statecharts formalism is considered a defacto standard for modeling reactive systems [1]. It describes how objects communicate and how they carry out their own internal behavior [2]. It is very often used for simulation, analysis, and code generation, making Statecharts modeling a critical activity in the development of software. As in any modern software development, statecharts (models) are developed incrementally following *stepwise refinement* strategies. For example, a developer starts by modeling a high-level Statecharts model, typically with no composite state, and then incrementally adds more detailed behavior in each state or transition. The Statecharts formalism supports incremental modeling by design through, for instance, the addition of orthogonal components in an AND-state or the replacement of a basic state with an OR-state. Each incremental modification, within the bounds of the rules defined in this paper, can be viewed as a *refinement* of the previous version. Refinement is the process of building a model gradually by making it more and more precise, while verifying that each refined model preserves the abstract behavior [3]. The advantage of starting with a more abstract model is that important properties can be defined in a simple model, which is therefore less likely to contain mistakes. Refinements then introduce detail in steps which are guaranteed to preserve the important properties.

There is a considerable amount of Statecharts refinement approaches proposed in the literature where the main concern is to preserve the behavior of the original model [3–8]. This is a necessary condition to refine a model. However, there is

* Corresponding author.
E-mail addresses: syriani@iro.umontreal.ca (E. Syriani), dasilvav@iro.umontreal.ca (V. Sousa), lucio@fortiss.org (L. Lúcio).

https://doi.org/10.1016/j.scico.2018.10.005
0167-6423/

almost no guarantee that the structure of the original model is preserved after refinement. This reduces the practicality of applying refinements in systems where Statecharts models are implementation artifacts. Developers manipulate them directly, thus the legibility of the design decisions must also be preserved.

The goal of this work is to enable developers to refine Statecharts models such that the preservation of the structure and behavior of the original statechart can be statically verified. This preservation helps with the incremental design of a statechart for future use, while minimizing the effect on existing systems using that statechart. Assuming the original model was correct, refining it at a specific location shall not affect the correctness of the rest of the model: all regression tests should still pass. Static verification is very useful in practice. It reduces the need to perform proofs and simulations at every refinement step as in other approaches [9]. From a tool perspective, it is also possible to offer rapid feedback to developers at design-time. Nevertheless, this requirement forces us to compromise among the various possibilities of refinements, by imposing restrictions over the permissible refinements that are guaranteed to preserve structure and behavior.

In this paper, we provide a set of rules that govern Statecharts refinement with the intention of preserving the *structure and reachability* of the original statechart. We allow refinement opportunities of Statecharts which permit the modeler to control refinements such that the resulting statechart is still compatible with the original. The refinements our approach allows must be useful in practice and enable the developer to control what refinements should be allowed in the future. Focusing on this kind of preservation has the potential to increase the predictability of the refinement steps, as well as respecting the design decisions made when it was initially developed. We consider the refinement process to be purely additive: i.e., no removal of elements from the original is allowed. One major advantage of the refinement technique we propose is that the preservation of the structure and behavior of the original statechart is guaranteed by construction: no model checking technique is needed. Refinement is not meant as a replacement for arbitrary modification, which follows an entirely distinct process that should be performed on the original statechart rather than on the refined one. These rules must also be focused on providing realistic and usable boundaries, so as not to constrain the modeler too much. From a practical point of view, our rules are best suited for a stepwise refinement development methodology that we illustrate on a real example.

This paper extends a preliminary work [10] significantly, with a formal validation of the refinement rules, the notion of structure and reachability preservation by refinement, a description of the complete implementation of the work and a discussion of the limitations of our approach. In Section 2, we define the notion of Statecharts refinement and its impact on the modeling process. In Section 3, we describe the rules governing the refinement for states and transitions. In Section 4, we formally prove the soundness of these rules and deduce useful properties. In Section 5, we describe a practical usage of Statecharts refinement with a concrete implementation. In Section 6 we illustrate the application and usability of the refinement on a simple example by incrementally modeling the behavior of an autonomous flying drone. In Section 7, we discuss the rationale behind our assumptions and decisions, and state some of the limitations of our approach. In Section 8, we discuss other refinement approaches and conclude in Section 9.

2. Formal definition of statecharts refinement

In this section, we present the Statecharts formalism at the level of abstraction needed for this work. Then we explain the need to preserve both structure and reachability when practically refining Statecharts and formally define this refinement relation.

2.1. Statecharts definition

As defined in [1], a Statecharts model is defined as a tuple $SC = \langle S, T, E, V, en, ex, in, I \rangle$. It consists of four sets of states $S = S_B \cup S_{OR} \cup S_{AND} \cup S_{pseudo}$, transitions T, events E, and variables V. E denotes both external events provided as stimuli from the context outside the statechart (the queue) and signals of internal events generated by the statechart. We consider time events and the null event φ just like any other event in E.

States can be basic S_B, OR S_{OR}, AND S_{AND}, or pseudo-states[1] (choice, fork, join, and history) $S_{pseudo} = S_{choice} \cup S_{fork} \cup S_{join} \cup S_{history}$. Basic states are primitive states that model an atomic execution. The system stays in a basic state until an outgoing transition is triggered. OR-states are composite states that encapsulate a substatechart. AND-states define orthogonal components where substatecharts can be executed in parallel, i.e., more than one state can be active at a time. Pseudo-states are control-flow nodes that are transient and that augment the expressiveness of transitions.

The relation $in \subseteq S \times S$ denotes the acyclic containment relationship which must be non-empty for OR-states and AND-states. Furthermore, history states are only contained in OR-states, i.e., $\forall s \in S,\ h \in S_{history}$ such that $(s, h) \in in$, then $s \in S_{OR}$. To illustrate the definition, consider the hierarchical statechart represented graphically in Fig. 1. At the root, it contains the OR-state X and basic state E. X contains two AND-states, one containing basic states A and B, and the other C and D.

A transition is a tuple $t = (s, e, x, g, s')$ where $s, s' \in S$ are the source and target states of the transition, $e, x \in E$ are the triggering and output events respectively and $g : \Xi \rightarrow \{true, false\}$ is a guard predicate for t that acts over the set of

[1] Although they are part of Harel Statecharts, we do not consider junctions and conditionals since they are simply syntactic shortcuts to reduce the number of transitions with similar enabling events and guards.

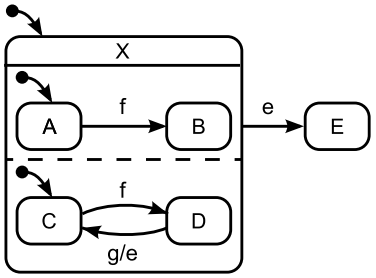

Fig. 1. An example of a statechart.

substitutions Ξ of variables V into their values. For convenience we write $var(g)$ to denote the variables of V that are used by guard g. Let $src, tar : T \to S$ denote the source and target state of a transition[2] respectively. Additionally, if $src(t) \in S_{fork}$ or $tar(t) \in S_{join}$, then $t = (src(t), \varphi, \varphi, true, tar(t))$ is unguarded i.e., $\forall \xi \in \Xi, g(\xi) = true$, and does not broadcast any event. Finally if $src(t) \in S_{choice}$, then t must be mutually exclusive with other transitions leaving $src(t)$. In Fig. 1, the transition from D to C is unguarded, triggered by event g, and it outputs event e.

As defined in [11], a configuration of a statechart is a maximally legal snapshot before or after a step. We formally define a configuration of SC as $(\bar{S}, \xi, \eta) \in \mathsf{CONF}_{SC}$, where $\bar{S} \subseteq S$, $\xi : V \to Value \in \Xi$ is a substitution function assigning values to the variables of the statechart, and η is the set of the last states that were active in all OR-states with history. $I = (S_\odot, \xi_\odot, \eta_\odot) \in \mathsf{CONF}_{SC}$ is the initial configuration representing the default states, initial value of variables, and initial history which consists of the default states of all OR-states containing a history pseudo-state. States can define entry and exit actions $en, ex : S \times \Xi \to \Xi$ which can modify the previous value of variables in V. In Fig. 1, $S_\odot = \{\mathsf{X}, \mathsf{A}, \mathsf{C}\}$. Although not illustrated in the figure, we suppose that each basic state has an entry action that prints the name of the state and an empty exit action. In this case, no variables are used.

We say that a transition $t = (s, e, x, g, s')$ is *enabled* for a configuration $(\bar{S}, \xi, \eta)$ where $s \in \bar{S}$, if $g(\xi) = true$ and e is the next event in the event queue. In case $e = \varphi$, only the fact that $g(\xi) = true$ enables t. When transition t is enabled for a configuration $(\bar{S}, \xi, \eta)$, we write $enabled(t, (\bar{S}, \xi, \eta))$. For example given the initial configuration in Fig. 1, if f is the first event in the queue, then the transitions outgoing from A and C are enabled. For our work, we rely on the Statecharts *outer-first step semantics* as defined in [11]: if two transitions are in conflict (enabled at the same time, but not in different orthogonal components of the same OR-state), then the transition outgoing from the outermost state of the currently active state hierarchy is the one enabled. In the example, the two transitions labeled with f are not in conflict. However, if the transition labeled with e were triggered by event f as well, then the three transitions would be in conflict. With outer-first semantics, only the latter would be enabled.

We also require the notion of sequential application of steps that consumes a list of events from a queue, which we denote *bigstep*. More formally, assume a *step* relation for statechart SC is defined such that $\xrightarrow{step} \subseteq \mathsf{CONF}_{SC} \times E \times \mathsf{CONF}_{SC}$. Let also Q_{SC} denote the set of all queues consisting of events from SC. We define the *bigstep* relation for SC as $\xrightarrow{bigstep} \subseteq \mathsf{CONF}_{SC} \times \mathsf{Q}_{SC} \times \mathsf{CONF}_{SC}$ to be the smallest bigstep relation satisfying:

$$\frac{}{(\bar{S}, \xi, \eta), \mathsf{Q}_\emptyset \xrightarrow{bigstep} (\bar{S}, \xi, \eta)}$$

$$\frac{(\bar{S}, \xi, \eta), e \xrightarrow{step} (\bar{S}', \xi', \eta'), \quad (\bar{S}', \xi', \eta'), Q \xrightarrow{bigstep} (\bar{S}, \xi, \eta)}{(\bar{S}, \xi, \eta), e :: Q \xrightarrow{bigstep} (\bar{S}, \xi, \eta)}$$

In this notation, $e :: Q \in \mathsf{Q}_{SC}$ is a queue having as first element $e \in E$. Also, $\mathsf{Q}_\emptyset$ denotes the empty queue. When executing a statechart, its input is the sequence of external events from the queue and the output consists of the signals produced by the entry and exit actions of states. New external events received are appended to the end of the queue. The input events the statechart reacts to and the output it produces in its actions are often referred to as the *external interface* of the statechart [8].

We briefly describe the $\xrightarrow{step} \subseteq \mathsf{CONF}_{SC} \times E \times \mathsf{CONF}_{SC}$ relation for a statechart SC, following the definition in [11]. When an event e is the first in the queue, it is removed from that queue and processed by the statechart. Assuming SC is in a configuration $c = (\bar{S}, \xi, \eta) \in \mathsf{CONF}_{SC}$, all non-conflicting transitions $t \in T$ enabled by the configuration are executed simultaneously. The execution of a transition $t = (s, e, x, g, s')$ entails the execution of the exit action of state s under substitution ξ, leading to an intermediate updated configuration $c'' = (\bar{S}, \xi'', \eta)$. The new state s' is entered leading to the execution of the entry actions of state s' which, in turn, updates configuration c'' to $c' = (\bar{S}', \xi', \eta')$ – where $S' = S''\backslash\{s\} \cup s'$,

[2] We do not consider hyperedges for transitions because their semantics is emulated with fork and join pseudo-states.

ξ' is a new substitution resulting from executing the exit actions of s' under substitution ξ' and η' is the updated history of the last exited states.

We illustrate the step semantics with the example in Fig. 1. Suppose that we provide the queue $Q = \langle \mathsf{f}, \mathsf{g} \rangle$. The initial configuration can produce the string AC after the execution of the entry actions of the default states. f being the first event in the queue, the transitions outgoing from A and C are enabled. The first *bigstep* completes by outputting the string BD. The queue now only consists of the event g, which initiates the second *step*. The statechart then produces the string C and event e is prepended to the queue. This entails a third *step* to produce the string E, which completes the second *bigstep*.

2.2. Preservation of structure and reachability

The refinement approach we propose abides to basic object-oriented (OO) design principles, such as the open-closed principle (OCP) stating that the model should be "open for extension, but closed for modification" [12], the Liskov substitutability principle (LSP) [13], and the principle of covariance and contra-variance [14]. Our premise is that the refined Statecharts model should utterly preserve the external interface and behavior as perceived by the entities it is collaborating with. All inputs accepted and all outputs produced from these inputs by the original model should be accepted and produced in the same order by the refined model. This means that external systems should still be able to send the same events to the refined statechart when it is in specific configurations. They also expect that during the *bigstep* execution, the refined statechart produces signals as expected from the original. To satisfy this post-condition, the reachability of configurations should be preserved. Masiero et al. [15] defined the notion of reachability for Statecharts in a way similar to Petri nets. A configuration is reachable if there is a sequence of events, starting from the initial configuration, that enables transitions between intermediate state configurations. Informally, **reachability preservation** means that, given a sequence of events in the external stimuli queue, every configuration reached in the original statechart shall be reached in the same order in the refined statecharts, where substates are allowed. Consequently, the refined statechart can at least receive the same queues of events and output at least the same signals as the original would. This is analogous to the principles of covariance and contra-variance between overridden methods in sub-classes in OO design, which is considered well-formed and safe. Reachability preservation is needed to make sure that any system interacting with the original statechart can still interact with the refined one, given the same queue, without requiring any modification, to conform to the LSP.

Furthermore, we also require that there is a mapping from the refined statechart to the original one by preserving states, transitions, and state hierarchy. This is needed because, although two statecharts may be behaviorally equivalent without sharing the same structure [16], the aim of the refinements we propose is tied to the practicality and usability in development, maintenance, and debugging activities. Informally, **structural preservation** means that any state in the original statechart must be present in the refined statechart, where the refined state may be composite. Also, for any transition connecting two states in the original statechart, there must be a path of transitions connecting the corresponding states in the refined statechart. Structure preservation abides to the OCP, which reduces future developments efforts on the model. Therefore, refinements must preserve the structure of the original statechart and the reachability of its configurations.

2.3. Refinement relation

Statechart refinement is a restricted form of modification of the original model such that design decisions are preserved in the refined model. Given a statechart SC, we use the notational abuse $t = s \xrightarrow{e[g]/x} s' \in SC$ to refer to a transition $t = (s, e, x, g, s')$ of SC. We also use the notation $s \xrightarrow{+} s' \in SC^+$ to refer to a path of at least one transition in the transitive closure of the transitions of SC. In the following definitions $SC_o = \langle S_o, T_o, E_o, V_o, en_o, ex_o, in_o, I_o \rangle$ and $SC_r = \langle S_r, T_r, E_r, V_r, en_r, ex_r, in_r, I_r \rangle$ denote the original and the refined statecharts, respectively.

Definition 1 *(Statecharts Refinement).* A statechart refinement is a triple (SC_o, R, SC_r), denoted $SC_o \overset{R}{\preceq} SC_r$, where $R \subseteq (S_o \cup T_o) \times (S_r \cup T_r)$ is a binary relation subject to the conditions that follow. The set of all Statecharts refinements in called $\Re$.

State and Transition Preservation

$$\forall st_r \in ST_r, \exists! st_o \in ST_o : (st_o, st_r) \in R$$
$$\text{where } ST_i = S_{B_i} \cup S_{OR_i} \cup S_{AND_i} \cup S_{pseudo_i} \cup T_i \text{ for } i = o, r \tag{1}$$

Hierarchy Preservation

$$\forall (s_o, s_r), (s'_o, s'_r) \in R,\ (s_o, s'_o) \in in_o \Rightarrow (s_r, s'_r) \in in_r^+$$
$$\text{where } in_r^+ \text{ is the transitive closure of } in \tag{2}$$

Event and Variable Preservation

$$E_o \subseteq E_r \wedge V_o \subseteq V_r \tag{3}$$

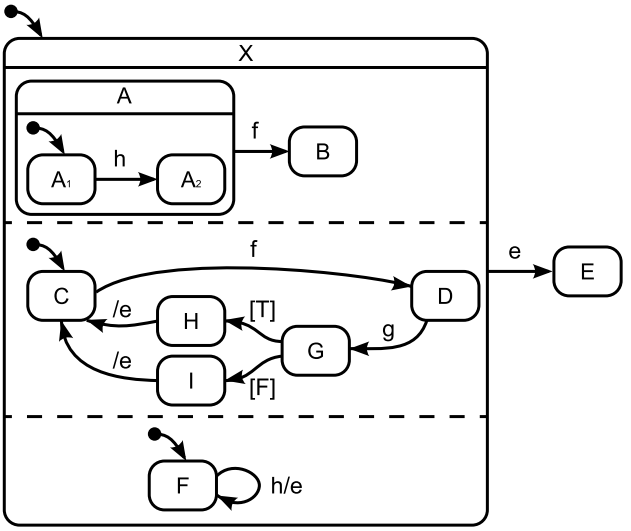

Fig. 2. A statechart refining the one in Fig. 1 according to Definition 1.

Connectivity Preservation

$$\begin{gathered}\text{if } \exists s_o \xrightarrow{e[g]/x} s' \in SC \text{ and } (s_o, s_r), (s'_o, s'_r) \in R \text{ then either } \exists s_r \xrightarrow{e[g]/x} s'_r \in SC_r \\ \text{or } \exists s_r \xrightarrow{e[g]} s''_r \xrightarrow{\varphi[true]/x} s'_r \in SC_r \text{ or } \exists s_r \xrightarrow{e[g]} s''_r \xrightarrow{+} s'''_r \xrightarrow{\varphi[true]/x} s'_r \in SC_r^+, \\ \text{where } (t_o, t_r) \in R \text{ and } t_o \in T_o, t_r \in T_r \end{gathered} \tag{4}$$

State Reachability Preservation

$$\begin{gathered}\forall Q_o, \in \mathbb{Q}_{SC_o}, \text{ if } I, Q_o \xrightarrow{bigstep} (\bar{S}, \xi, \eta) \\ \text{then } \exists Q_r \in \mathbb{Q}_{SC_r} \text{ such that } Q_o \overset{R}{\preceq} Q_r \text{ and } I_r, Q_r \xrightarrow{bigstep} (\bar{S}_r, \xi_r, \eta_r), \\ \text{where } \forall s \in \bar{S}, s_r \in \bar{S}_r, (s, s_r) \in R \end{gathered} \tag{5}$$

In the notation used above, we define $Q_o \overset{R}{\preceq} Q_r$ to mean that Q_o is a subsequence of Q_r, imposing that the order of the events in Q_o is preserved in Q_r.

Let us now informally describe the refinement relation above. Condition (1) ensures that any state or transition in SC_r is mapped back to exactly one state or transition in SC_o. Therefore the inverse refinement R^{-1} is a surjection. Condition (2) ensures that state hierarchy is preserved. Condition (3) ensures that all original events and variables must be preserved in the refined statechart. A consequence of Condition (4) is that if there is a transition from s_o to s'_o in SC_o, then the refinement can either preserve this transition or there must be a SC_r, where s_r is the refined version of s_o and s'_r is the refined version of s'_o. In this path, the transition outgoing from s_r must preserve the original event and guard, and the transition incoming to s'_r must have no guard or event and output the original output event. Through transitivity we can preserve the connectivity of any path defined in SC_o. Condition (5) guarantees that states that are reachable during execution can still be reachable after refinement. When the above conditions are met, we say that SC_r preserves the *structure and reachability* of SC_o.

Symbolic analysis of entry and exit actions is outside the scope of this paper, therefore we do not enforce a constraint between ξ_o and ξ_r. Nevertheless, because we rely on the identity to refine all elements of the statechart that are not explicitly refined in a rule, we assume that the original entry and exit actions of every state in the original statechart are preserved in the refined states. With this assumption, the sequence of all outputs produced by the original statechart shall be preserved by the refined statechart.

Consider the statechart in Fig. 2 that is a possible refinement of the one in Fig. 1. We notice that every state and transition path is preserved. For instance, state E remains identical. Basic state A is still present but is now an OR-state containing two substates. OR-state X now contains an additional AND-state. States D and C are still connected by a cyclic path of transitions. It is clear that Conditions (1)–(4) are satisfied, therefore the structure of the original statechart is preserved.

To verify the reachability preservation of the refined statechart, we can see that, given any queue, the trace output by the original statechart (the names of basic states entered) is a subsequence of the one output by the refined statechart, where substates are allowed. For example, given the queue $Q = \langle f, g \rangle$, the original statechart outputs the sequence AC BD C E and the refined statechart outputs A_1CF BD G H C E. The output of the original statechart is then a subsequence of the output of

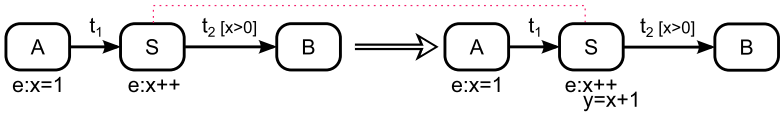

Fig. 3. Refinement of a state into a state.

the refined statechart, a constraint that is implied[3] by Condition (5). Nonetheless, this statechart allows for new behaviors that were not possible in the original statechart. For instance, if we provide the queue $Q = \langle h \rangle$ to the refined statechart, it will output: A_1CF A_2CF E. This shows that the refined statechart allows for more behaviors than the original, while preserving the original behavior. Consequently, the Statecharts refinement defined in Definition 1 allows for the following possible modifications: new events can be interjected between events from the original (i.e., new steps are possible), new states can be added, and new transitions can be added. However, these additions are subject to specific constraint rules in order to reflect the intention of predictive refinement in the context of incremental modeling.

3. Refinement rules for statecharts

In the previous section, we laid out the requirements for refinement rules. In this section, we define the rules that the developer is bound to, so he can produce a well-formed refined statechart.

In the definitions that follow we assume an original statechart to be refined $SC = \langle S, T, E, V, en, ex, in, I \rangle$ and the statechart resulting from the refinement $SC_r = \langle S_r, T_r, E_r, V_r, en_r, ex_r, in_r, I_r \rangle$. Refinement rules are additive: SC_r is obtained by introducing a new statechart $\hat{SC} = \langle \hat{S}, \hat{T}, \hat{E}, \hat{V}, \hat{en}, \hat{ex}, \hat{in}, \hat{I} \rangle$ in SC. For each refinement rule, $\hat{SC}$ is subject to constraints that serve as precondition before applying a specific rule. All rules are intended to be statically verified, while constraints specify assumptions on the dynamic behavior of the statechart.

3.1. Rules for state refinement

Constraint 1 *(Constraint on actions).* $\forall s_r \in S_r, \forall \xi \in \Xi$, let $\xi' = en_r(s_r, \xi)$. Then if $\forall t = (s, e, x, g, s') \in T, \exists \xi'' \in \Xi$ such that $g(\xi'') = false$ where $s, s' \in S$ and $e, x \in E$, then $\forall v \in var(g)$ we have that $\xi'(v) = \xi(v)$. The same holds for $\xi' = ex_r(s_r, \xi)$.

This constraint states that the new entry and exit actions of states in SC_r cannot modify the values of any variable $v \in V$ if v is used to restrict a guard of a transition in SC. Nevertheless, refined entry and exit actions can modify newly introduced variables $v_r \in V_r \setminus V$. This is needed to ensure the refinement does not prevent the enabling of an already existing transition.

Rule 1 *(Action rule).* Let SC be a statechart and $s \in S_B \cup S_{OR} \cup S_{AND}$ be a state of SC. Let also $\hat{SC} = \langle \{s\}, \emptyset, \emptyset, \hat{V}, \hat{en}, \hat{ex}, \emptyset, \emptyset \rangle$ be another statechart that satisfies Constraint 1. Refining entry and exit actions of state s produces a statechart SC_r such that: $S_r = S$, $T_r = T$, $E_r = E$, $V_r = V \cup \hat{V}$, $en_r = en \cup \hat{en}$, $ex_r = ex \cup \hat{ex}$, $in_r = in$ and $I_r = I$.

This rule adds further action in the entry and exit actions of s, possibly modifying values of newly introduced variables. As seen in Fig. 3, the statechart remains structurally the same, but the refined state has an additional action that does not manipulate a variable used in an existing guard. The refinement keeps a mapping between the original state and the updated state, as denoted by the red doted line.

Constraint 2 *(Constraint on broadcast).* $\forall t_r = (s_r, e_r, x_r, g_r, s'_r) \in T_r \setminus T$, if $x_r \neq \varphi$ then either $e_r \neq \varphi$ or $\exists v_r \in V_r, \exists \xi \in \Xi$ such that $g_r(\xi(v_r)) = false$.

This constraints states that newly introduced transitions cannot broadcast an event unless they require a non-null event to enable them or have a guard that may disable them for some configuration. This is needed in an outer-first semantics setting in order to prevent existing states to be unreachable in SC_r that were previously reachable in SC. To illustrate this situation, suppose that, in the statechart SC_r in Fig. 2, the transition between A_1 and A_2, enabled by event h, outputs the event e. If we provide the queue $Q = Q_r = \langle f, e \rangle$, SC outputs BD E and SC_r outputs BD E. Thus B is still reachable after refinement. However, if the transition between A_1 and A_2 is now enabled by φ, B is no longer reachable with any Q_r.

[3] To fully demonstrate that Condition (5) is preserved by the refinement we would have to extensively show the condition holds for all input queues. This is impossible due to the fact that queues can be arbitrarily large. Rather, we demonstrate formally in Appendix A that if the modeler only makes use of the refinement rules we propose in Section 3, then Condition (5) always holds for any refinement and input queue.

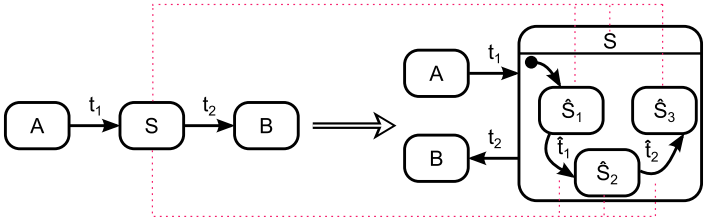

Fig. 4. Refinement of a basic state into an OR-state.

Rule 2 *(Basic-to-OR state rule).* Let SC be a statechart, $s \in S_B$ be a basic state of SC, and $\hat{SC}$ be another statechart where all of its components are disjoint from the corresponding ones in SC. Assume $\hat{SC}$ satisfies Constraints 1–2. Refining state s into an OR-state containing statechart $\hat{SC}$ produces a statechart SC_r such that:

1. $S_{r_B} = S_B \setminus \{s\} \cup \hat{S}_B$, $S_{r_{OR}} = S_{OR} \cup \{s\} \cup \hat{S}_{OR}$, $S_{r_{AND}} = S_{AND} \cup \hat{S}_{AND}$ and $S_{r_{pseudo}} = S_{pseudo} \cup \hat{S}_{pseudo}$
2. $T_r = T \cup \hat{T}$, $E_r = E \cup \hat{E}$, $V_r = V \cup \hat{V}$, $en_r = en \cup \hat{en}$, $ex_r = ex \cup \hat{ex}$
3. $in_r = in \cup \bigcup_{\hat{s} \in \hat{S}} (s, \hat{s}) \cup \hat{in}$ where $\hat{s}$ is not contained in any other state
4. $I_r = \begin{cases} (S_\odot \cup \hat{S}_\odot, \xi_\odot \uplus \hat{\xi}_\odot, \eta_\odot \cup \hat{\eta}_\odot) & \text{if } s \in S_\odot \\ (S_\odot, \xi_\odot \cup \hat{\xi}_\odot, \eta_\odot) & \text{otherwise} \end{cases}$ where $I = (S_\odot, \xi_\odot, \eta_\odot)$ and $\hat{I} = (\hat{S}_\odot, \hat{\xi}_\odot, \hat{\eta}_\odot)$[4]

Fig. 4 illustrates the refinement of a basic state into an OR-state as per Rule 2. The statechart SC is on the left-hand side and the SC_r is on the right-hand side. The refined state is removed from the set of basic states and added to the set of OR-states of the original statechart. All the states of statechart $\hat{SC}$ become part of the newly created OR-state. The default states of $\hat{SC}$ become part of the default state of the original statechart SC whenever state s being refined is default. A consequence of having $\hat{SC}$ disjoint from SC is that new states in $\hat{S}$ and transitions in $\hat{T}$ only work on local variables in $\hat{V}$, i.e., the new ones. This also creates a mapping R, between the corresponding elements of SC and SC_r, as per Definition Rule 1. The state s is mapped to the new OR-state s, along with all elements of $\hat{SC}$, as they are introduced in the refinement of s. In Fig. 4 the dotted lines only shows the refinement mapping R for the new elements in SC_r, since all elements of SC_r are mapped by their identity in SC.

Rule 3 *(OR-to-AND state rule).* Let SC be a statechart, $s \in S_{OR}$ be an OR-state of SC, and $\hat{SC}$ be another statechart where all of its components are disjoint from the corresponding ones in SC. Assume $\hat{SC}$ satisfies Constraints 1–2. Let $T_{in} = \{t \mid src(t) = s\}$ and $T_{out} = \{t \mid tar(t) = s\}$. Refining state $s \in S_{OR}$ into an AND-state containing statechart $\hat{SC}$ produces a statechart SC_r such that:

1. $S_{r_B} = S_B \cup \hat{S}_B$, $S_{r_{OR}} = S_{OR} \cup \hat{S}_{OR} \cup \{s_{or}\}$, $S_{r_{AND}} = S_{AND} \cup \hat{S}_{AND} \cup \{s_{and}\}$ and $S_{r_{pseudo}} = S_{pseudo} \cup \hat{S}_{pseudo}$ such that $s_{or} \notin S_{OR} \cup \hat{S}_{OR}$ and $s_{and} \notin S_{AND} \cup \hat{S}_{AND}$
2. $T_r = T \cup \hat{T} \cup \{(s_{and}, \hat{e}, \hat{x}, s') \mid \exists (s, e, x, s') \in T_{in}$ where $\hat{e} = e, \hat{x} = x$ for some $s' \in S\} \cup \{(s', \hat{e}, \hat{x}, s_{and}) \mid \exists (s', e, x, s) \in T_{out}$ where $\hat{e} = e, \hat{x} = x$ for some $s' \in S\} \setminus T_{in} \setminus T_{out}$
3. $E_r = E \cup \hat{E}$, $V_r = V \cup \hat{V}$, $en_r = en \cup \hat{en}$, $ex_r = ex \cup \hat{ex}$
4. $in_r = in \setminus \{(p, s)\} \cup \{(s_{and}, s), (s_{and}, s_{or}), (p, s_{and})\} \cup \bigcup_{\hat{s} \in \hat{S}} (s_{or}, \hat{s}) \cup \hat{in}$ where $\hat{s}$ is not contained in any other state and $p \in S$
5. $I_r = \begin{cases} (S_\odot \cup \hat{S}_\odot, \xi_\odot \uplus \hat{\xi}_\odot, \eta_\odot \cup \hat{\eta}_\odot) & \text{if } s_{or} \in S_\odot \\ (S_\odot, \xi_\odot \cup \hat{\xi}_\odot, \eta_\odot) & \text{otherwise} \end{cases}$ where $I = (S_\odot, \xi_\odot, \eta_\odot)$ and $\hat{I} = (\hat{S}_\odot, \hat{\xi}_\odot, \hat{\eta}_\odot)$

Fig. 5 illustrates the refinement of an OR-state into an AND-state as specified in Rule 3. In SC_r on the right-hand side, there is a new AND-state S_{AND} that contains all the original OR-states as well as a new OR-state containing $\hat{SC}$. Note that step 2 makes sure all transitions incoming to s (e.g., t_1) and all transitions outgoing from s (e.g., t_2) are rerouted so that S_{AND} is their source or target, respectively. The remaining elements of SC_r remain unchanged otherwise. In this case the R in Fig. 5 maps $\hat{SC}$, the created AND-state and OR-state alongside $\hat{SC}$ and s_r to state s.

Rule 4 *(AND-state rule).* Let SC be a statechart, $s \in S_{AND}$ be an AND-state of SC, and $\hat{SC}$ be another statechart where all of its components are disjoint from the corresponding ones in SC. Assume $\hat{SC}$ satisfies Constraints 1–2. Refining state $s \in S_{AND}$ into an AND-state containing statechart $\hat{SC}$ in an additional region produces a statechart SC_r such that:

1. $S_{r_B} = S_B \cup \hat{S}_B$, $S_{r_{OR}} = S_{OR} \cup \hat{S}_{OR} \cup \{s_{or}\}$, $S_{r_{AND}} = S_{AND} \cup \hat{S}_{AND}$ and $S_{r_{pseudo}} = S_{pseudo} \cup \hat{S}_{pseudo}$ such that $s_{or} \notin S_{OR} \cup \hat{S}_{OR}$

[4] The $\uplus$ operator enforces that values coming from the variable substitutions on the right of the operator override their counterparts on the left of the operator, thus guaranteeing the resulting substitution is still a function.

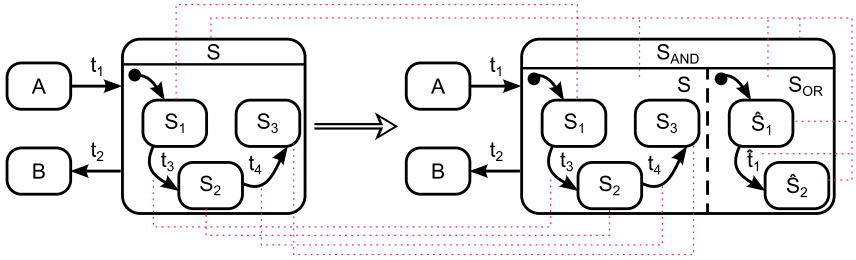

Fig. 5. Refinement of an OR-state into an AND-state.

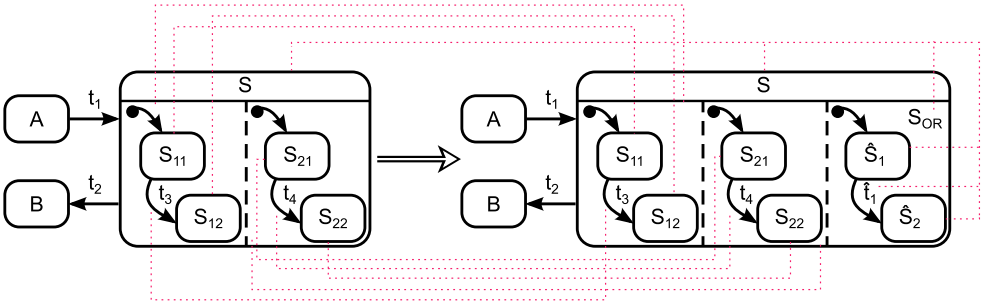

Fig. 6. Refinement of an AND-state into an AND-state.

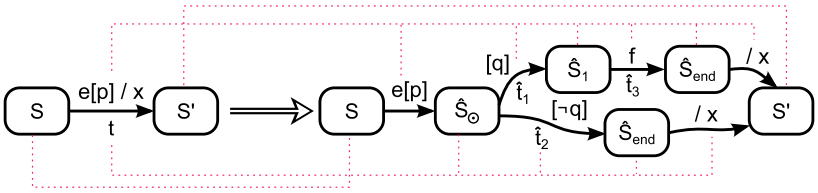

Fig. 7. Refinement of a transition.

2. $T_r = T \cup \hat{T},\ E_r = E \cup \hat{E},\ V_r = V \cup \hat{V},\ en_r = en \cup \hat{en},\ ex_r = ex \cup \hat{ex}$
3. $in_r = in \cup \{(s, s_{or})\} \cup \bigcup_{\hat{s} \in \hat{S}} (s_{or}, \hat{s}) \cup \hat{in}$ where $\hat{s}$ is not contained in any other state
4. $I_r = \begin{cases} (S_\odot \cup \hat{S}_\odot, \xi_\odot \uplus \hat{\xi}_\odot, \eta_\odot \cup \hat{\eta}_\odot) & \text{if } s \in S_\odot \\ (S_\odot, \xi_\odot \cup \hat{\xi}_\odot, \eta_\odot) & \text{otherwise} \end{cases}$ where $I = (S_\odot, \xi_\odot, \eta_\odot)$ and $\hat{I} = (\hat{S}_\odot, \hat{\xi}_\odot, \hat{\eta}_\odot)$

Fig. 6 illustrates the refinement of an AND-state into an AND-state as per Rule 4. The new elements of $\hat{SC}$ are encapsulated in a new OR-state s_{or} in the refined statechart SC_r. R maps s_{or} and all of $\hat{SC}$ to s, the original state to refine.

3.2. Rules for transition refinement

In the following, we denote, for the new statechart to introduce $\hat{SC}$, $\hat{S}_\odot \subseteq \hat{S}$ as the set of states that are not contained in any other state and $\hat{s}_\odot \in \hat{S}_\odot \cap \hat{S}^I$, where $(\hat{S}^I, \hat{\xi}^I, \hat{\eta}^I) = \hat{I}$. We also denote $\hat{S}_{end} \subseteq \hat{S}$ where any $\hat{s}_{end} \in \hat{S}_{end}$ is not the source of any transition in $\hat{T}$ and is not contained in any other state.

Constraint 3 *(Constraint on paths).* $\exists Q \in \mathbb{Q}_{\hat{SC}}, \hat{s}_{end} \in \hat{S}_{end}$ such that $\hat{I}, Q \xrightarrow{bigstep} (\{\hat{s}_{end}\}, \hat{\xi}, \hat{\eta})$.

This constraint states that the newly introduced statechart $\hat{SC}$ must have at least one state $\hat{s}_{end}$ without outgoing transitions that must be reachable from the default state. An example is depicted in Fig. 7. The constraint ensures that s' will still be reachable. Although this is a strong constraint, this is only needed for transition refinement rules.

Rule 5 *(Transition rule).* Let SC be a statechart containing a transition $t = s \xrightarrow{e[g]/x} s' \in T$ where $s \in S \setminus (S_{fork} \cup S_{history})$ and $s' \in S \setminus (S_{fork} \cup S_{join})$. Let $\hat{SC}$ be another statechart where all of its components are disjoint from the correspond-

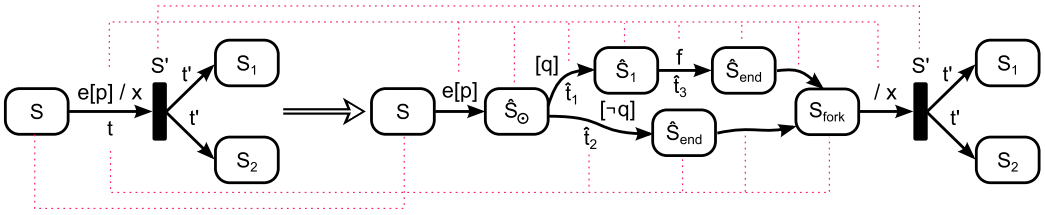

Fig. 8. Refinement of a transition to a fork.

ing ones in SC and that satisfies Constraints 1–3. Refining transition t into statechart $\hat{SC}$ produces a statechart SC_r such that:

1. $S_r = S \cup \hat{S}$
2. $T_r = T \setminus \{t\} \cup \hat{T} \cup \{s \xrightarrow{e[g]/\varphi} \hat{s}_\odot\} \cup \bigcup_{\hat{s}_{end} \in \hat{S}_{end}} \{\hat{s}_{end} \xrightarrow{\varphi[true]/x} s'\}$
3. $in_r = \begin{cases} in \cup \hat{in} & \text{if } s \text{ and } s' \text{ do not have any parent} \\ in \cup \bigcup_{\hat{s} \in \hat{S}_\odot} (LCP(s, s'), \hat{s}) \cup \hat{in} & \text{otherwise} \end{cases}$
   where $LCP(s, s')$ is the least common parent state containing s and s'
4. $E_r = E \cup \hat{E},\ V_r = V \cup \hat{V},\ en_r = en \cup \hat{en},\ ex_r = ex \cup \hat{ex},\ I_r = I$

Fig. 7 illustrates the refinement of a transition t from s to s' into a set of states and transitions. The first transition outgoing from s retains the original event and guard in the refined statechart. All incoming transitions to s' must broadcast the original event x. All intermediate states in $\hat{SC}$ must be connected such that there is always a path from s to s'. R maps all these states and transitions to t. Regarding the containment of the states added by the refinement, if either the source s or target s' of t is not contained in any other state, then nor are the states of $\hat{SC}$. If s or s' are contained in the same state, then all states of $\hat{SC}$ are as well. Otherwise, s or s' have a different parent, so the states of $\hat{SC}$ are contained in the least common parent of the s and s'.

Rule 6 *(Fork rule).* Let SC be a statechart containing a transition $t = s \xrightarrow{e[g]/x} s' \in T$ where $s \in S \setminus (S_{fork} \cup S_{history})$ and $s' \in S_{fork}$. Let $\hat{SC}$ be another statechart where all of its components are disjoint from the corresponding ones in SC and that satisfies Constraints 1–3. Refining transition t into statechart $\hat{SC}$ produces a statechart SC_r such that:

1. $S_{r_B} = S_B \cup \hat{S}_B \cup \{s_{fork}\},\ S_{r_{OR}} = S_{OR} \cup \hat{S}_{OR},\ S_{r_{AND}} = S_{AND} \cup \hat{S}_{AND}$ and $S_{r_{pseudo}} = S_{pseudo} \cup \hat{S}_{pseudo}$ such that $s_{fork} \notin S_B \cup \hat{S}_B$
2. $T_r = T \setminus \{t\} \cup \hat{T} \cup \{s \xrightarrow{e[g]/\varphi} \hat{s}_\odot\} \cup \bigcup_{\hat{s}_{end} \in \hat{S}_{end}} \hat{s}_{end} \xrightarrow{\varphi[true]} s_{fork} \cup \{s_{fork} \xrightarrow{\varphi[true]/x} s'\}$
3. $in_r = \begin{cases} in \cup \hat{in} & \text{if } s \text{ and } s' \text{ do not have any parent} \\ in \cup \bigcup_{\hat{s} \in \hat{S}_\odot} (LCP(s, s'), \hat{s}) \cup \hat{in} & \text{otherwise} \end{cases}$
4. $E_r = E \cup \hat{E},\ V_r = V \cup \hat{V},\ en_r = en \cup \hat{en},\ ex_r = ex \cup \hat{ex},\ I_r = I$

Fig. 8 illustrates the refinement of a transition as per Rule 6. This is the same rule as for transition refinement in Rule 5, with the following additions. We add an unguarded transition with no output from each ending state in $\hat{S}_{end}$ to a new state s_{fork}. We also add an unguarded transition outputting the same output x as t from s_{fork} to the fork. s_{fork} is needed since a fork can only have one incoming transition. R defines the same mapping as in Rule 5.

3.3. Rules for refinement by extension

Constraint 4 *(Constraint on transitions).* Let $\hat{t}_{start} = (s, \hat{e}, \hat{x}, \hat{g}, \hat{s}_\odot)$ be a transition with $s \in S$, $\hat{e}, \hat{x} \in \hat{E}$ and $\hat{s}_\odot \in \hat{S}_\odot \cap \hat{I}$ as in Section 3.2. If $\exists s' \in S\ \forall t = (s^+, e, x, g, s') \in T$ where $s^+ \in s \cup in^+(s)$ and $\hat{e} = e$, then $\forall \xi \in \Xi, g(\xi) \rightarrow \neg \hat{g}(\xi)$.

This constraint states that for a given configuration ξ, $\hat{t}_{start}$ cannot be enabled if there is another transition with the same source state that can be enabled with ξ, such as t in Fig. 9. This is necessary to ensure that the resulting statechart SC_r is still deterministic. Furthermore, this restriction also applies to any transitions inside s when s is an OR-state or an AND-state. This ensures that pre-existing segments of the statechart don't become inaccessible as a result of the outer first policy.

Rule 7 *(State extension rule).* Let SC be the statechart we are refining. Let $\hat{SC}$ be another statechart where all of its components are disjoint from the corresponding ones in SC. Let $s \in S \setminus (S_{join} \cup S_{history})$ be the state we want to refine and

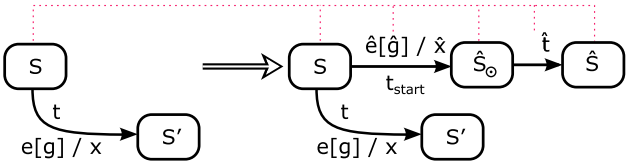

Fig. 9. Refinement by extending a state with a statechart.

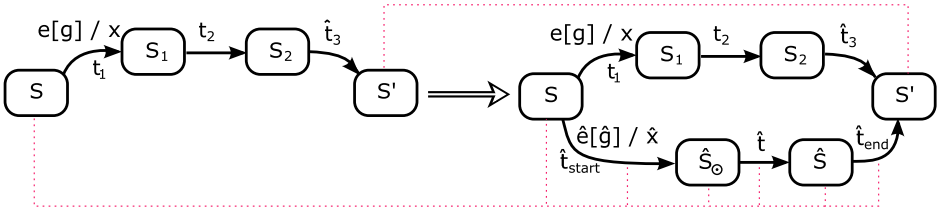

Fig. 10. Refinement by extending a new path between two states.

$\hat{t}_{start} = (s, \hat{e}, \hat{x}, \hat{g}, \hat{s}_\odot)$ be a new transition restricted by Constraint 4. Refining s by extending it with statechart $\hat{SC}$ produces a statechart SC_r such that:

1. $S_r = S \cup \hat{S}$
2. $T_r = T \cup \hat{T} \cup \{\hat{t}_{start}\}$
3. $in_r = \begin{cases} in \cup \bigcup_{\hat{s} \in \hat{S}_\odot} (p, \hat{s}) \cup \hat{in} & \text{if } \exists p \in S \mid (p, s) \in in \\ in \cup \hat{in} & \text{otherwise} \end{cases}$
4. $E_r = E \cup \hat{E},\ V_r = V \cup \hat{V},\ en_r = en \cup \hat{en},\ ex_r = ex \cup \hat{ex},\ I_r = I$

Fig. 9 illustrates a state extension refinement rule as per Rule 7. This rule allows one to create a new transition $\hat{t}_{start}$ from an existing state s to a new statechart $\hat{SC}$ with states that were not in SC. Constraint 4 guarantees that $\hat{t}_{start}$ does not conflict with another transition to keep SC_r deterministic. We then define the mapping R, from all elements of $\hat{SC}$ and the new transition to s.

Rule 8 *(Path refinement rule).* Let SC be the statechart we are refining. Let $\hat{SC}$ be another statechart where all of its components are disjoint from the corresponding ones in SC. Let $s \xrightarrow{+} s' \in SC^+$ be the path starting from the state $s \in S \setminus (S_{join} \cup S_{history})$ we want to refine and $\hat{t}_{start} = (s, \hat{e}, \hat{x}, \hat{g}, \hat{s}_\odot)$ be a new transition restricted by Constraint 4. We also define another new transition $\hat{t}_{end} = (\hat{s}, \varphi, x, g, s')$ for some $\hat{s} \in \hat{S}$, such that the path $\exists \hat{s}_\odot \xrightarrow{+} \hat{s} \in \hat{SC}^+$. Refining the path from s to s' with statechart $\hat{SC}$ produces a statechart SC_r such that:

1. $S_r = S \cup \hat{S}$
2. $T_r = T \cup \hat{T} \cup \{\hat{t}_{start}\} \cup \{\hat{t}_{end}\}$
3. $in_r = \begin{cases} in \cup \hat{in} & \text{if } s \text{ and } s' \text{ do not have any parent} \\ in \cup \bigcup_{\hat{s} \in \hat{S}_\odot} (LCP(s, s'), \hat{s}) \cup \hat{in} & \text{otherwise} \end{cases}$
4. $E_r = E \cup \hat{E},\ V_r = V \cup \hat{V},\ en_r = en \cup \hat{en},\ ex_r = ex \cup \hat{ex},\ I_r = I$

Fig. 10 illustrates a path refinement as per Rule 8. In this refinement we extend the ways we can reach state s' from state s without affecting the applicability of the preexisting paths between these two states. For this purpose, the transition t_{start} from s to the starting state $\hat{s}_\odot$ of $\hat{SC}$, should not conflict with any outgoing transition from s such as t_1, as in the rule of state extension in Rule 7. The purpose of adding t_{end} from a state in $\hat{SC}$ to s' is to ensure that s' is reachable from this new path. R maps all new elements in SC_r that were not in SC to s.

3.4. Refinement of pseudo-states

All pseudo-state refinements are the identity, i.e., they cannot be refined. This is also the case for transitions outgoing from a history state or a fork, and incoming to a join.

4. Formal rule application

We are interested in applying sequences of refinements to an initial statechart, in order to reach a final design. It is thus of the utmost importance to be able to show that sequences including applications of *refinement rules*, defined in Section 3 are statechart refinements, as formally stated in Definition 1. Refinement rules can be applied in isolation in any order.

This section provides the formal foundation and necessary proofs of the soundness of our approach. The reader who is more interested by the implementation and application of our approach may safely skip this section without any prejudice to the understanding of the remaining of the paper. In what follows, we will only present the headings of our formal definitions and properties, and refer the reader to Appendix A for the mathematical proofs. More precisely, the main results of this section are the proofs that refinement rules are formally statechart refinements, as per Definition 1. We will then go on to formally prove that sequences of refinements are also refinements.

Let us then start by formally demonstrating that the refinement rules from Section 3 are refinements. We start by building *rule applications* consisting of a triple including only the part of the statechart being refined by the rule, the part of the statechart created by the rule, and a traceability relation between the two. The notion of *local rule application* captures only how the part of the statechart affected by these rules is changed.

Definition 2 *(Local rule application).* Let statechart SC_r be the result of applying a *refinement rule* to a statechart SC.
An application of a state refinement rule (Rules 1–4) or a state extention rule (Rule 7) to a state s of SC is a triple (SC, R, SC_r), where R is a binary relation between s and each new state or transition created by the state refinement rule in SC_r. The five kinds of state refinement rules imply local rule applications as follows:

- *Basic State* (Rule 1): the local rule application (SC, R, SC_r) is such that $R = \{(s, s)\}$.
- *Basic-to-OR State* (Rule 2): the local rule application (SC, R, SC_r) is such that $R = \{(s, s)\} \cup \{(s, s_r) \mid s_r \in S_r \setminus S\} \cup \{(s, t_r) \mid t_r \in T_r \setminus T\}$.
- *OR-to-AND State* (Rule 3), *AND-to-AND State* (Rule 4) and *State Extension* (Rule 7), the local rule application (SC, R, SC_r) is such that $R = \{(s, s)\} \cup \{(s, s_r) \mid s_r \in S_r \setminus S\} \cup \{(s, t_r) \mid t_r \in T_r \setminus T\} \cup R_{lid}$ where R_{lid} is the identity relation for states contained in s united with the identity relation for transitions between states contained in s.

A local application of a transition refinement rule (Rules 5–6) on a transition t of SC is a triple (SC, R, SC_r), where $R = \{(t, s_r) \mid s_r \in S_r \setminus S\} \cup \{(t, t_r) \mid t_r \in T_r \setminus T\} \cup R_{lid}$ and R_{lid} is the identity relation for the source and target states of t united with all states and transitions contained within them.
The local application of the *Path Refinement Rule* (Rule 8) on $s \xrightarrow{*} s'' \in SC^*$, is a triple (SC, R, SC_r), with $R = \{(s, s_r) \mid s_r \in S_r \setminus S\} \cup \{(s, t_r) \mid t_r \in T_r \setminus T\} \cup R_{lid}$ where R_{lid} is the identity relation of the s and s'' states united with all states and transitions contained within them.

Figs. 4–10 depict examples of state, transition, and path refinement local rule applications. The respective mapping R for each rule is depicted by the red dotted lines.

4.1. Soundness

The following lemma ensures that the changes made locally by state, transition, and path refinement rules are formal refinements that satisfy the conditions of Definition 1.

Lemma 1 *(Soundness: local rule application is a (local) refinement). Refinement rules* create (local) refinements restricted to the elements of the statechart being refined and satisfy the five refinement conditions. More formally, given a statechart SC, and a local rule application (SC_l, R, SC_{lr}), we have that $SC_l \overset{R}{\preceq} SC_{lr} \in \Re$.

Intuitively, a statechart can be refined to itself. The following property ensures that the refinement relation is reflexive. This is an important result we will use subsequently.

Property 1 *(Refinement reflexivity).* For any statechart SC we have that $SC \overset{R_{id}}{\preceq} SC \in \Re$, where R_{id} is the union of the identity relations for states and transitions of SC.

It follows from Property 1 that the *copy* operation of a statechart is a refinement: this is the identity refinement. The following defines that the application of a refinement rule is a local rule application *merged* with the identity refinement. This allows us to add to the local refinement all elements of the statechart untouched by the rule and the remaining traceability relations, thus rebuilding the complete application of the rules. Note that the merge operation "glues" the local rule application on top of the identity refinement, while removing the replaced elements resulting from the identity refinement.

Definition 3 *(Rule application).* The application of a *state refinement* rule or a *state extension* rule to a statechart SC on state s consists of a merge of the identity refinement $id = SC \overset{R_{id}}{\preceq} SC$ and a local refinement $loc = SC \overset{R_{loc}}{\preceq} SC_r$ of a state s of SC. This is denoted $merge(id, loc) = SC \overset{R}{\preceq} SC_r$ where $R = R_{loc} \cup R_{id} \setminus \{(s, s)\}$ and SC_r is the statechart obtained from applying the refinement rule.
Similarly, the application of a *transition refinement* rule to a statechart SC on a transition t is also a merge of the identity refinement and a local refinement of a transition t of SC, denoted $merge(id, loc) = SC \overset{R}{\preceq} SC_r$ where $R = R_{loc} \cup R_{id} \setminus \{(t, t)\}$.
Finally, the application of a *path refinement* rule to a statechart SC on a path between two states s and s'' is also a merge of the identity refinement and a local refinement of a transition t of SC, denoted $merge(id, loc) = SC \overset{R}{\preceq} SC_r$ where $R = R_{loc} \cup R_{id} \setminus \{(s, s), (s'', s'')\}$.

The following finally shows that a rule application is a refinement by proving that the result of merging an identity refinement with a local rule application refinement of a statechart as done in Definition 3 is still a refinement.

Lemma 2 *(Rule application is a refinement).* Let id be the identity refinement and loc be a local rule application of a *refinement rule*. Then, the rule application $merge(id, loc) = SC \overset{R}{\preceq} SC_r \in \Re$ is a refinement.

Having shown that applying state, transition, or path refinement rules result in formal refinements, we now show that sequences of formal refinements are refinements. We do so by showing the transitivity of statechart refinement in Property 2, which naturally leads to our final result in Property 3.

Property 2 *(Refinement transitivity).* If $SC \overset{R_1}{\preceq} SC_{r_1} \in \Re$ and $SC_{r_1} \overset{R_2}{\preceq} SC_{r_2} \in \Re$, then there exists a refinement $SC \overset{R_3}{\preceq} SC_{r_2} \in \Re$.

Property 3 *(Sequence of rule applications is a refinement).* The composition of two or more rule applications $SC_1 \overset{R_1}{\preceq} SC_2 \in \Re, \ldots, SC_{n-1} \overset{R_{n-1}}{\preceq} SC_n \in \Re, \forall n \geq 3$, denoted $SC_1 \overset{R_1}{\preceq} SC_2 \overset{R_2}{\preceq} \cdots \overset{R_{n-1}}{\preceq} SC_n$ is a refinement.

Therefore, conceptually, every rule application is a valid refinement and a sequence of rule applications is also a refinement. This enables modelers to define multi-step refinements and hence allows for incrementally refining statecharts.

4.2. Completeness

Thanks to Property 3, it is possible to define new rules that encapsulate specific combinations of some of the eight rules presented, thus allowing further expressiveness for refinement. For example, it is possible to connect two states that are not (transitively) connected by first applying Rule 7 followed by Rule 8. Additional rules can be added to the eight we propose, as long as they satisfy the refinement definition and can not be achieved by composing existing rules or overlap with them. We discovered this specific set of rules from the ground up, trying to achieve refinements on several case studies. This set of rules is not complete, but provides guidelines to perform refinements that are useful in practice while preserving the structure and behavior.

5. Implementation

Part of the value of this work lies in its concern of practicality. Refinement can be verified statically so that a tool can assist developers who want to refine their Statecharts models. We have implemented Statecharts refinement in the modeling tool AToMPM [17]. In this section we describe how the tool is to be used as well as some implementation details.

5.1. Process

In our tool, the process of refining Statecharts is broken into two major steps: the application of a refinement rule, followed by the validation that the refinement is correct.

5.1.1. Original statechart

The developer first constructs the original statechart model or opens an existing one in AToMPM. Once this design of the statechart is complete and well-formed, the refinement process can begin. The developer initiates the *refine* command within the tool, which changes AToMPM from *Edit* mode to *Refine* mode.

5.1.2. Element selection

Once in *Refine* mode, the tool restricts all editing operations, only allowing selections on existing Statecharts models. In this way, the developer is able to select the element to be refined without compromising the guaranties provided by statechart refinement. Only in the case of path refinement, the developer must select two states. An additional check verifies that the two states are indeed connected by a path.

5.1.3. Rule selection

When the developer selects the element to refine, the tool proposes a list of refinement rules to choose from. Only rules that are compatible with the selected element are proposed. For example, if he selects a basic state to be refined, only Rules 1, 2, and 7 will be proposed.

5.1.4. Refinement

When the desired rule is selected, the tool opens a new statechart editor. The developer then constructs the local refinement statecharts.[5] He can either create the statechart from scratch or load an existing one and then edit it accordingly. In this step, the tool automatically generates a mapping between every refined or created element and the selected element from the original statechart.

5.1.5. Validation

When the developer is satisfied with the refinement statechart model, he initiates the *validate* command which verifies that the model is a valid refinement and saves it. The developer can save an invalid model at anytime during the process to complete it later. This step ends when the validation is successful.

The *validate* command verifies that the selected rule is properly applied on the selected element. It verifies that the refinement satisfies Constraint 1 when applying Rule 1, Constraints 1–2 are when applying Rules 2–4, Constraints 1–3 when applying Rules 5–6, and Constraint 4 when applying Rules 7–8. This command therefore ensures a correct refinement and notifies the developer of any constraint violated for the selected rule.

Note that, for practical reasons, although all eight refinement rules are proven to be correct by construction (see Appendix A), a validation of the refinement is still necessary because the developer has the freedom to create invalid refinements that do not satisfy Definition 1. Another advantage of implementing an explicit validator is that, given two arbitrary statecharts, the validator can be used to determine whether one is a refinement of the other.

5.1.6. Refined statechart

When no failure is reported, a new refined statecharts is created. This is performed by a model transformation that copies all elements of the original statechart and replaces the refined element by the local refinement statechart model he built. This is a straightforward transformation thanks to the mappings created at the refinement step.

5.2. Design

Fig. 11 illustrates an excerpt of the overall design of our implementation. The `AToMPM` class holds triggers for both the refine and validate commands. Each of these forwards the request to the respective `Validator` or `Refiner` classes, that handle type processing and developer prompting where needed. The `Refiner` contains one instance of each subclass of `BaseRefiner`. Each subclass implements the protected methods of `BaseRefiner` following the Template design pattern. The refiners are responsible for creating the mappings from the original to the refined statechart, though mappings could conceivably be made manually between the two statecharts and the `MappingValidator` will still determine whether or not the mappings are correct.

The `Validator` iterates through all of the mappings and evaluates all of the applied constraints until the list is exhausted. This will result in `SUCCESS` if all constraints evaluate to `SUCCESS`. The result is `WARN` if some constraints evaluate to `WARN`. A `WARN` result indicates that the static verification passes, but that further semantical analysis of actions and guards is required. These are situations where the developer should be careful to ensure that the action code satisfies the assumptions on the dynamic behavior of the statechart, as stated in the Constraints 1–4. These situations are discussed in Section 7.1. The result is `FAIL` if at least one constraint evaluates to `FAIL`. In the latter two cases, a list of all warning and failed constraints are displayed to the developer along with their description. Note that we do not need to verify the refined statechart for well-formedness since the conformance check of the statechart model with the Statecharts metamodel is already taken care of by AToMPM. Each `Constraint` object holds the id of the original model element and the necessary data it needs to verify that the corresponding constraint holds. The `evaluate` operation verifies that the mapping satisfies the specific constraint it encodes. This class follows the Command pattern and is meant to allow easy addition of new, fully functional, self-contained constraints. The evaluation of each `Constraint` object returns a `ConstraintResult` as a result of their evaluation, acknowledging the satisfaction of the constraint. A `NONE` result indicates that the constraint does not apply to elements evaluated. In the following, we provide implementation details of the refiner and the validator.

[5] This corresponds to $\hat{SC}$ in the rules of Section 3.

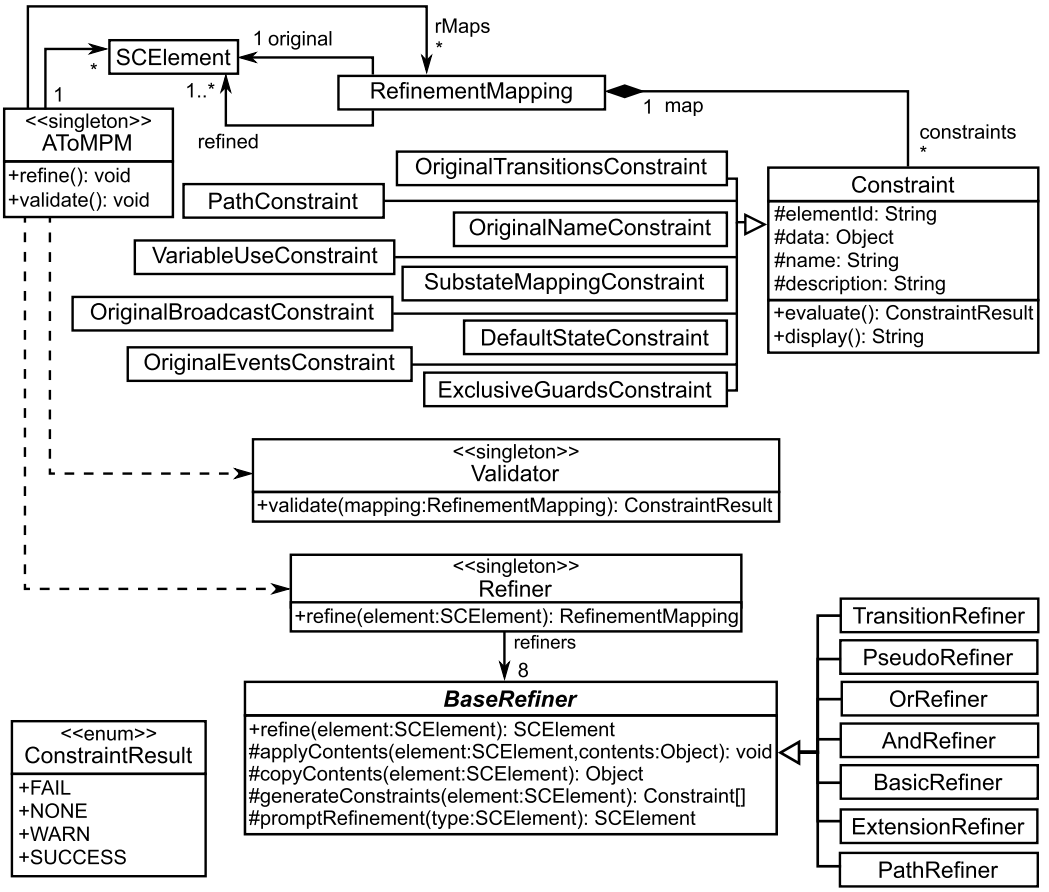

Fig. 11. The class diagram for the refiner and validator.

5.2.1. Refiner

The developer is first prompted to select the element(s) to be refined and has to choose the type of refinement to be performed on the selected elements. For example, if he selects an `OR-state`, the prompt asks him to choose from the `Action Rule`, `OR-to-AND Rule` or `State Extension Rule`. For any rule chosen, the following steps are always executed:

1. Create a new local statechart $\hat{SC}$ in a separate editor.
2. When the modification of this statechart is complete, the constraints associated with the selected rule validate the refinement. By default, the OriginalName, OriginalTransitions and BroadcastConflict constraints are always added.
3. When no fail message remains and all warnings are cleared, $\hat{SC}$ is merged with a copy of the original statechart to form SC_r.

On top of this default behavior, we add further pre-processing operations on the local statechart and constraints specific to each rule. We refer to the notation used in Section 3 in what follows.

- Rule 1: $\hat{SC}$ consists of a copy of the selected state s and its actions. The constraint VariableUse is added to the validator.
- Rule 2: $\hat{SC}$ consists of a copy of the selected state s converted to an empty OR-state. When merging $\hat{SC}$ in SC_r, all transitions adjacent to s are recreated with the same events, guards and broadcasts, as adjacent to the new state s_r.
- Rule 3: $\hat{SC}$ consists of an empty OR-state s_{OR}. The constraint SubstateMapping is added to the validator. When merging $\hat{SC}$ in SC_r, a new AND-state s_{AND} will contain two orthogonal components: s_{OR} and a copy of s. All transitions adjacent to s are recreated in the copied statechart SC_r with the same events, guards and broadcasts, as adjacent to s_{AND}.
- Rule 4: $\hat{SC}$ consists of an empty OR-state s_{OR}. Merging consists of adding $\hat{SC}$ as an orthogonal component of s.
- Rules 5–6: $\hat{SC}$ is an empty statechart. The constraint ExclusiveGuards is added to the validator. When merging $\hat{SC}$ in SC_r, $s_{\odot}$ is no longer marked as default. A new transition is created between $src(t)$, t being the selected transition, and $s_{\odot}$ with the original event and guard of t. Additional transitions are created between each state in $\hat{SC}$ with no outgoing transitions and $tar(t)$ with the original broadcast of t.

- Rule 7: $\hat{SC}$ consists of a copy of the selected state s. Any modification to this state will not be taken into account during the merge. The constraint ExclusiveGuards is added to the validator. The path in $\hat{SC}$ starting from the new outgoing transition t_{start} from s is merged in SC_r.
- Rule 8: This is the only case where two states of SC are selected. Before $\hat{SC}$ is created, the Path constraint is used to verify that a path connects the two states. $\hat{SC}$ consists of only a copy of the two states s and s'. Any modification to these states will not be taken into account during the merge. Constraints ExclusiveGuards and Path are added to the validator. The path in $\hat{SC}$ starting from the new outgoing transition t_{start} from s until the new incoming transition t_{end} to s' is merged in SC_r.

5.2.2. Validator

The validator ensures that the resulting statechart is a valid refinement with respect to Definition 1. We briefly summarize the `evaluate` operation of each `Constraint` subclass. In the current implementation, we support the following constraint validations:

- OriginalName: searches through the named `SCElements` within a refinement mapping for any element that matches the original element's name (e.g., state and event names). This validates the preservation of named elements in all rules.
- OriginalTransitions: searches through all states within a refinement mapping validating each state's input and output transitions against the original ones. This validates the original transition preservation in all rules.
- BroadcastConflict: searches through new transitions within in a refinement mapping. No event broadcast is allowed on intermediate transitions unless the trigger event is not null. This is more permissible than Constraint 2 since we cannot statically verify that the guard is neither a tautology or a contradiction. Consequently, we let the user make sure that such guards are not always true or always false.
- ExclusiveGuards: searches through all resulting transitions within a refinement mapping of the application of Rules 5–8. If any guarded transition is found, it verifies that a literal negation of that guard is present on another transition with the same event trigger from the same source state, or any of its substates. All guard validations are currently done at a literal level and do not check for logically equivalent expressions. Furthermore, broadcast events on these mutually exclusive transitions must be different.
- Path: given two states, it verifies that there is a structural path from the former to the latter by performing a depth-first search.
- SubstateMappings: searches through all of the states within a refinement mapping where the original `SCElement` is a composite state. For every original substate, we validate that there exists a mapping to a refined substate. This validates the substate mapping preservation rule defined for AND and OR-states in Rules 3–4.
- DefaultState: searches within a refinement mapping through all states that are marked as default in the original statechart and verifies that their refined counterpart is also marked as default. This validates Rules 1–4.
- VariableUse: for every variable used in actions introduced by the refinement, it checks that these variables are not used on any existing guard.

6. Incremental statechart modeling

The refinement approach we present in this paper is best suited when the development follows a stepwise refinement strategy. To illustrate the usability of the rules governing the refinement of Statecharts, we present a series of refinements that showcase the use of the refinement rules in sequence and as alternatives for modeling the behavior of drones.

6.1. Drone modeling example

When modeling a statechart with the purpose of allowing possible refinements in the future, special care should be invested in the initial model. In this example, we are seeking to model the behavior of simple autonomous quadcopters, a.k.a. drones.

6.1.1. Specifying the original statechart

The basic specification is to support the following sequence of drone operation: start with the drone completely disabled, start up the system, take off, fly around, land, and shut down when the flight is over.

Consider the initial statechart in Fig. 12 that models a generic behavior for drones. Each state in SC.Original acts as a placeholder of the minimal phases we require for the behavior of any drone. Note that we encapsulate the Take Off, Fly, Descend and Landed states inside an OR-state Operational. This enables us to provide future refinements that collectively affect all four states of operation.

The overall behavior is a simple loop. The drone is initially in Off state with all its electrical and mechanical system turned off. We assume that events are received from an external queue corresponding to requests from a remote controller. Upon receiving an on event, the startup() function is executed as entry action of the Start state, which turns the electrical and mechanical systems on.

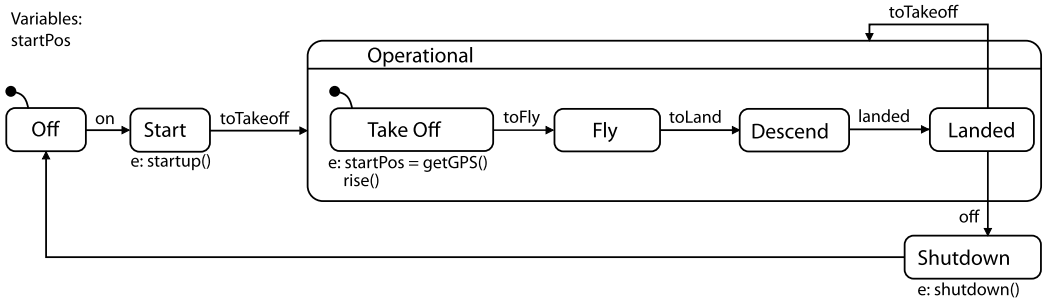

Fig. 12. SC.Original: Generic drone behavior.

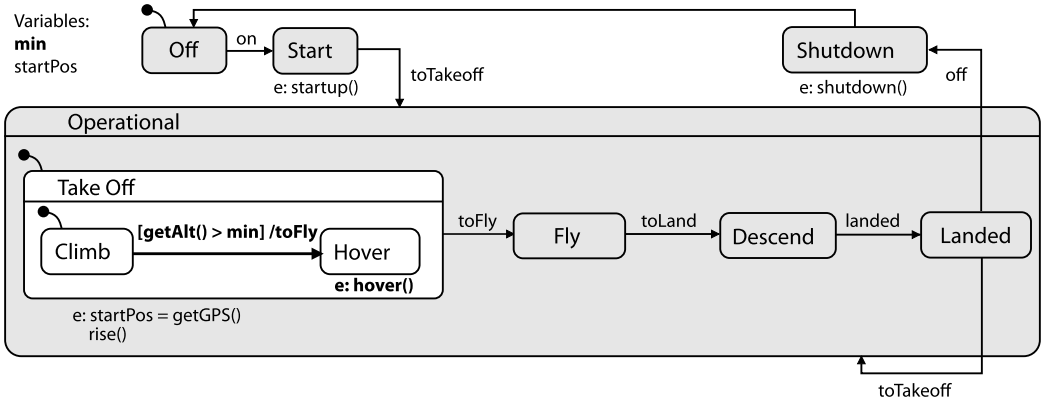

Fig. 13. SC.TR1: result after refining take off using Rule 2.

The drone goes into operational mode when it receives a toTakeoff event. Before rising in the air, the entry action of the Take Off state initializes the starting position variable to the current GPS coordinates. It is the only variable used in the statechart. In our setting, we assume that functions to move the drone will stay active until an event explicitly stops them, e.g., rise(). For example, in state Take Off, the drone will keep on rising unless an event stopping it is raised, such as toFly. When in the Fly state, it can perform any flight function: e.g., movements in any x–y-z direction. Upon receiving a toLand event, the drone goes down while in the Descend state. When the drone reaches solid ground, we assume a landed event is issued, which transitions to the Landed state. Note that flying and descending are underspecified purposely to not limit refinement opportunities. Had we defined a descend() operation in the entry action of Descend, we would not be able to customize the landing process as it would be the first action to perform. Defining it in the exit action would be semantically incorrect because it would only be executed when landed is received. Therefore, we opted to remain more abstract and not specify any action for this initial version. Moreover, transition specification also has an impact on future refinements. For example, the detection of solid ground could have been modeled as a guard, but this would have (1) limited the use of the position variable in new state actions and (2) obviated the possibility for explicitly representing a sensor for ground detection in the statechart.

While landed, the drone has two options. If it receives a toTakeoff event, the operation restarts from taking off. Otherwise, if it receives an off event, the shutdown() procedure is initiated. Upon completion, the statechart goes back instantaneously to the Off state with a null transition. Note that the structure of the statechart imposes certain rules regarding the order of the operations. For example, the only way to exit the Operational state (i.e., shutting down) is from the Landed state (i.e., when the drone is on the ground). This ensures a certain safety so that the drone cannot be shutdown mid-flight, as that would lead to hazardous consequences.

6.1.2. Adding detailed behavior

This initial statechart is very simplistic and provides the basic generic behavior a drone should implement. To be more realistic, we refine different modes of the drone's operation by adding more detailed behavior.

Safe take off We may want to refine how the drone takes off the ground before flying. For example, we refine the Take Off state so the drone rises until a minimum safety altitude is reached. This refinement is performed by applying Rule 2 to SC.Original by transforming the basic state into an OR-state that includes a more detailed statechart. The resulting statechart is illustrated in Fig. 13, denoted Takeoff Refinement 1 (SC.TR1). The drone will rise until its altitude has reached Min and

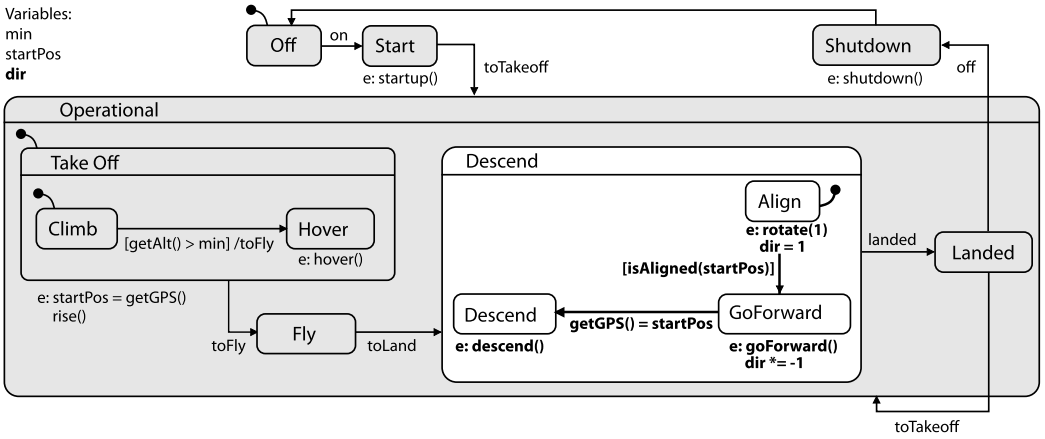

Fig. 14. SC.LR1: result after refining descend applying Rule 2 on SC.TR1.

then stay still thanks to the hover() operation. We included the transition to trigger toFly directly, but could have also relied on an external event to trigger it.

In the following figures of this section, we highlight states directly affected by refinements in white (e.g., Take Off) and newly introduced variables, operations and transitions in bold (e.g., min). The elements in gray remain unaltered. For simplicity, we do not include formal proof of correct rule applications here. All refinements have been performed using the prototype presented in Section 5 that conservatively implements all refinement rules. Informally, we can see that this application is correct since it does not affect already existing variables and none of the broadcasts introduced occur unguarded.

Safe landing In practice, it is possible for the drone to receive a toLand event when it is flying over an area where it cannot safely land. To solve this, we specify a landing procedure so that it searches for a safe place before landing. We do so by refining the Descend state to align the drone with its takeoff point and fly straight back to it. We consider it a safe place to land because we know there was no obstacle there during its ascension. We apply this refinement to SC.TR1 by using Rule 2, resulting in the statechart Landing Refinement 1 (SC.LR1) depicted in Fig. 14.

6.1.3. *Refining by concern*

When a concern is not included in the original statechart, it is possible to add a behavior specific to one concern using refinement. We can add such concern without modifying existing statechart components, but it may still have a side-effect on them.

Battery control The previous statechart model assumed infinite battery capacity. However, this concern must be addressed if we want to operate a physical drone. Therefore, we apply a series of refinements to regulate the behavior of the drone in case of low battery.

Sufficient energy concerns all modes of operation. Thanks to the fact that they are all encapsulated in an OR-state, the first refinement is to add an orthogonal component that monitors the battery charge and issues a warning in case of low battery. For this purpose, we use Rule 3, to transform the Operational OR-state into an AND-state, resulting in the statechart Battery Refinement 1 (SC.BR1) depicted in Fig. 15.

When the battery is critically low, just a warning is not enough in the case of autonomous flight. Therefore, we need to increase the granularity of the battery monitoring to take this into consideration. For this second refinement, we can simply refine the guarded transition that initiates the warning using Rule 5. The resulting statechart triggers the toLand event if the battery is critically low, in addition to issuing the warning. The resulting statechart is Battery Refinement 2 (SC.BR2). The new structure of the transition refinement is depicted in the bottom part of Fig. 16. Note the use of the choice pseudo-state (diamond shape) used to refine the guard. To ensure reachability, whenever a guard is introduced during a transition refinement, a mutually exclusive guard alternative must be provided to make sure at least one of the transitions will be enabled and the target state is reached. This is enforced by the validator in our implementation.

The battery can already be critically low during takeoff. However, SC.BR2 does not handle that situation yet. We therefore apply a path refinement using Rule 8 to bypass the Fly state and directly go into Descend. This ensures us that the drone does not gain too much altitude when its battery is critically low, thus increasing the risk of the drone falling. Fig. 16 shows the statechart Battery Refinement 3 (SC.BR3) after applying Rule 8 to SC.BR2.

Notice that we could have refined the critically low battery monitoring first: directly having the guard to 10% and triggering the toLand event as shown in SC.BR1b in Fig. 17. However, doing so would not have allowed us to perform a subsequent refinement to issue a warning before landing. The problem is that transition refinement forces the original guard to be set on the first intermediate transition to preserve the original behavior. Therefore, adding a later transition

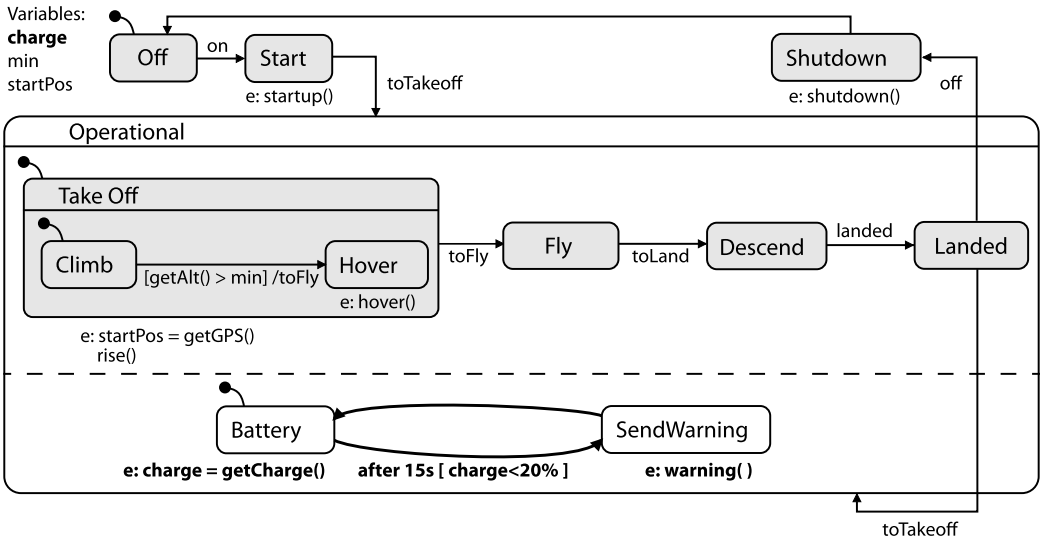

Fig. 15. SC.BR1: result after refining operational applying Rule 3 on SC.TR1.

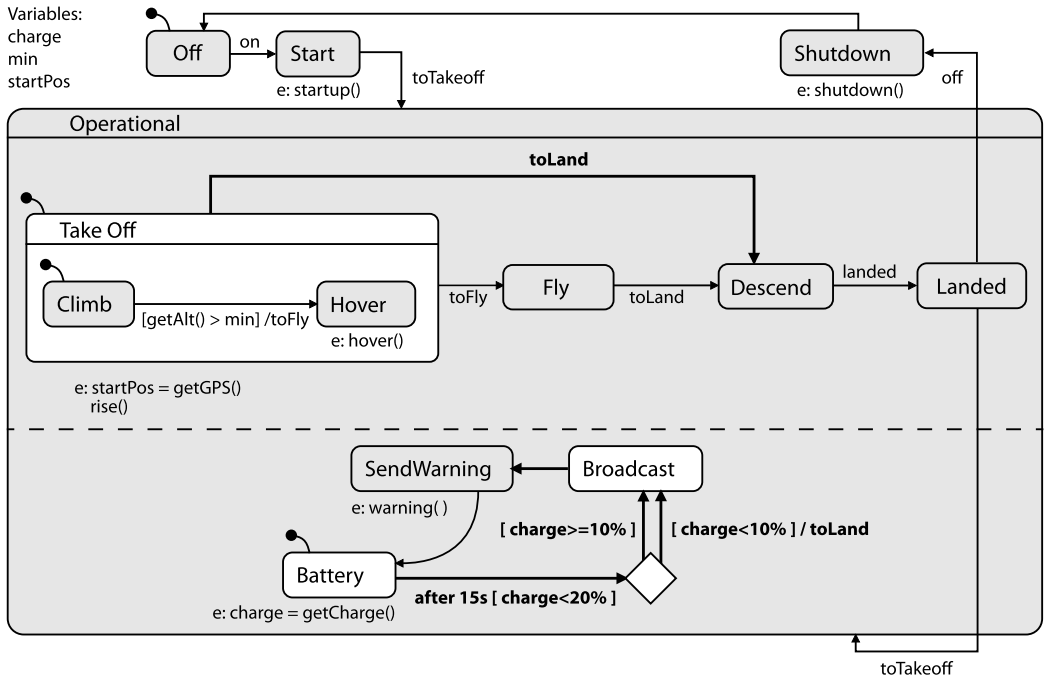

Fig. 16. SC.BR3: land when critically low and abort rising by applying Rule 5 on SC.BR1 followed by Rule 8.

with a weaker guard of 20% would not produce the desired behavior. This shows that the order of rule application has important consequences that may prevent further desired refinements.

Takeoff exceptions Some special circumstance can occur during takeoff: an obstacle (such as a tree) may prevent the drone from reaching its minimal altitude. We can address this concern in a modular way by adding the required sensor on the drone. Let us take SC.TR1 as base statechart. We apply Rule 3 which adds an orthogonal component that models sensing obstacles so it can coordinate with the remaining statechart. When an obstacle is detected, the transition issues a cancel event. The resulting statechart after this first refinement is Takeoff Refinement 2 (SC.TR2).

We now need the statechart to react accordingly when an obstacle is detected. Like with *Battery control*, we want the drone to move directly to the Descend state. For this we apply the same path refinement from Takeoff to Descend with Rule 8 as we did previously. The new transition relies on the toLand event to be triggered. The resulting statechart is Takeoff Refinement 3 (SC.TR3).

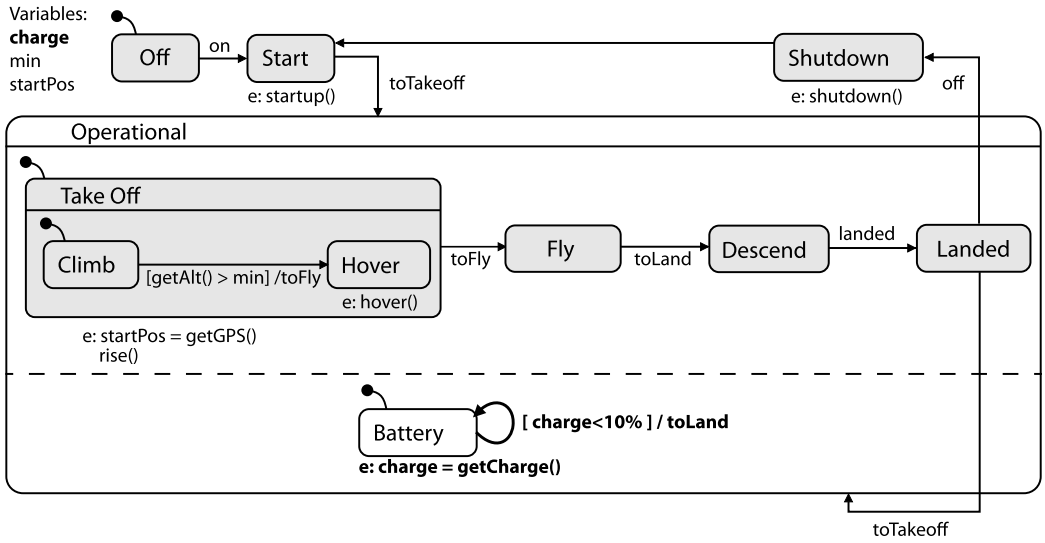

Fig. 17. SC.BR1b: land battery critically low by applying Rule 5 to SC.TR1.

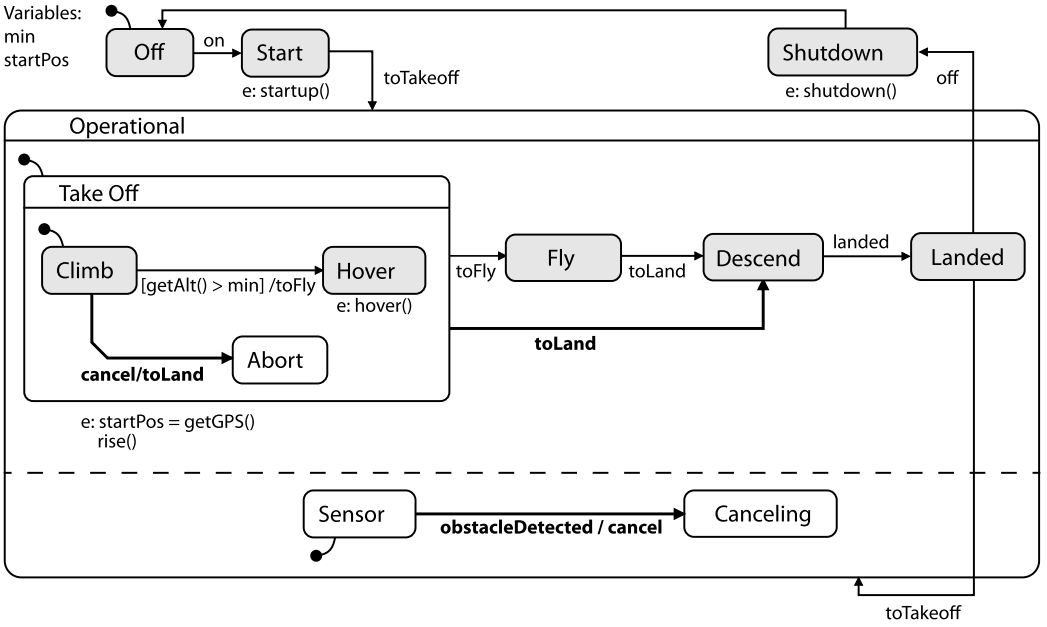

Fig. 18. SC.TR4: handle obstacle during take off by applying Rule 3 on SC.TR1 followed by Rule 8 and Rule 7.

The use of the toLand event to trigger the new path creates compatibility with other refinements by concern. However, in this case, this event is not broadcast directly from the sensor component. To deal with this, we make a third refinement to convert the cancel event raised by the sensor into a toLand event. This should be done in the Take Off state to react while rising. For this purpose, we use Rule 7 that extends the Rise state with an alternative transition that broadcasts the toLand event. This resulting statechart Takeoff Refinement 4 (SC.TR4) is presented in Fig. 18.

6.1.4. Delaying guards

The final set of refinements focuses on specifying detailed behavior of how the drone flies while in the Fly state. The first refinement is to define a general flying procedure specific to flying in a loop. For that, we start from SC.TR1 and apply Rule 2 to transform the Fly state into an OR-state. Looking at Fig. 19, we add eight states for moving forward, turning, and holding in place. The resulting statechart Fly Refinement 1 (SC.FR1) depicts the general structure of a family of algorithms for flying in a loop.

One specific flight procedure in this family is to perform an S-shaped path back and forth to, for example, scan a rectangular field. We therefore refine SC.FR1 by using Rule 1 consecutively on each Turn state. This adds entry and exit

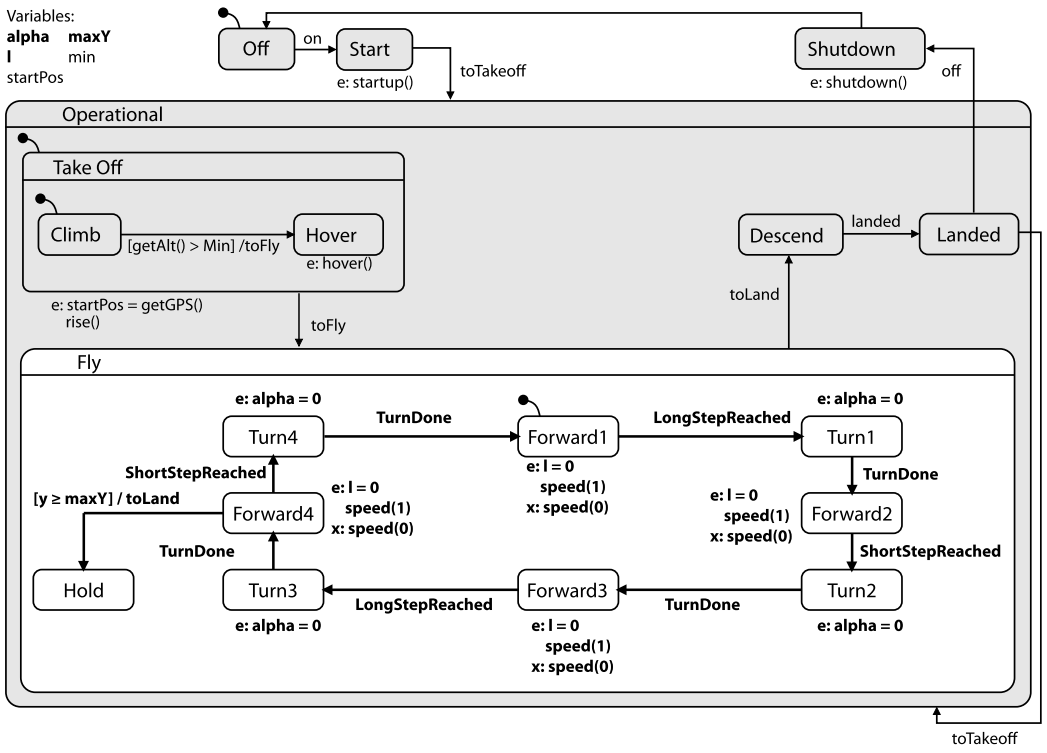

Fig. 19. SC.FR1: General fly in loop algorithm by applying Rule 2 on SC.TR1.

actions to rotate the drone continuously clockwise rotate(1), counter-clockwise rotate(-1), or to stop rotating rotate(0). This function modifies the value of the direction angle alpha of the drone. The resulting statecharts are called Fly Refinement 2 to Fly Refinement 5 (SC.FR5). Notice how the first refinement did not specify these actions to remaining generic and allow refinements for other loop-based flight paths. Also, Rule 1 is safe to apply on SC.FR1 since the variable alpha is never used in a guard yet.

The only thing missing to make the drone fly as desired is to decide on the timing of each step. We model this by adding an orthogonal component that encodes the condition over the position and angle variables for each step. To that end, we apply Rule 3 once to convert the Operational state into an AND-state, followed by consecutive applications of Rule 4 to add additional orthogonal components. Each new orthogonal component should make sure that the occurrence of the events broadcast are in a specific order. This leads to statecharts Fly Refinement 6 to the final Fly Refinement 13 (SC.FR13) which is illustrated in Fig. 20.

Had we defined these conditions as guards directly in the Fly state, we would not have been able to define any operation over the used variables. Instead, we used events to serve as placeholders. Events are then broadcast by the orthogonal components. This refinement strategy allows for more modularity and reduces limitations of other future refinements.

6.2. Refinement evolution tree

As demonstrated in the drone example, Statecharts refinement is to be used in an evolutionary process. This also illustrates the need for multi-step refinements defined in Property 3. After each refinement, we obtain a fully functional statechart that can be used in an application. Refinements are not mandatory, but are thought of extensions of the current implementation. To have an overview of the refinements applied, we can build a *refinement evolution tree*, presented in Fig. 21 for the drone example. Nodes represent statecharts and edges represent refinement rule applications. The root is the original statechart.

Some properties of the evolution tree are visible with the rendered layout. Along the vertical axis (depth of the tree), refinements correspond to the addition of more detailed behavior as in Section 6.1.2. Along the horizontal axis (breadth of the tree), refinements correspond to the addition of new concerns or alternative designs, as in Section 6.1.3. In Fig. 21, we see that SC.TR1 is used as base statechart to spawn different behaviors of drones for different requirements. Similarly, SC.FR1 can be used to spawn the behavior of different looping drones and SC.FR5 to spawn the behavior of different S-shape scanning drones. We also note that some refinements can be applied across multiple branches of the tree, such as the extension of the path to land using Rule 8. Other types of refinements that can also be re-used are those that have no

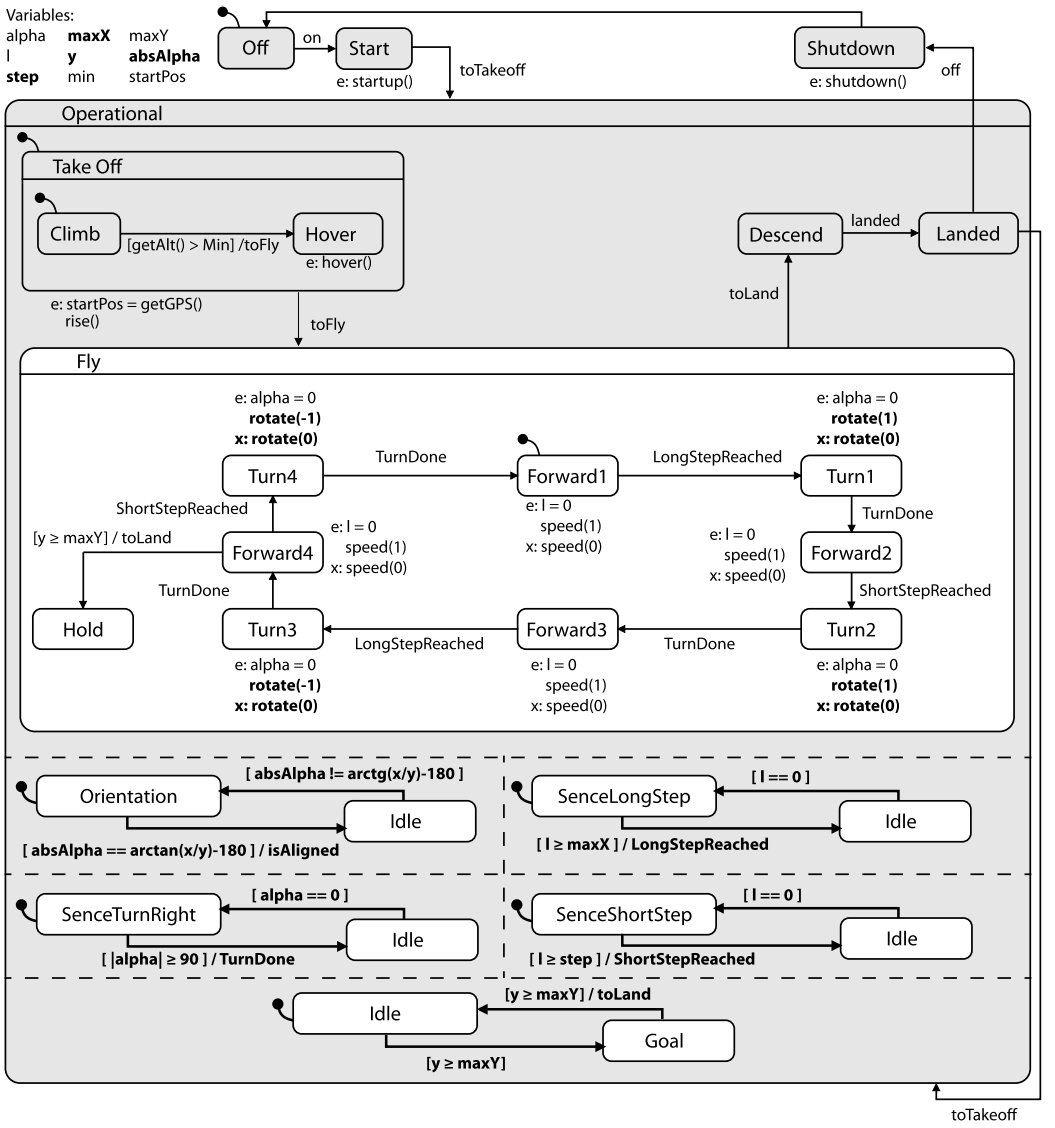

Fig. 20. SC.FR13: S-shaped back and forth flight pattern by applying four times Rule 1 on SC.FR1, followed by Rule 3 and four times Rule 4.

side-effect on the statechart outside the scope of its local application. For example, the refinement from SC.TR1 to SC.LR1 to refine landing can, as a matter of fact, be used on all statecharts in sibling branches.

In Section 3, we presented the refinement rules grouped syntactically with respect to the element that is refined in the original statechart. With the refinement evolution tree of a specific example, we can also group the rules according to the purpose of the refinement. Rules 1, 2, 5, and 6 are mainly used to add more precise detail to a more generic behavior. These are rules that either expand the behavior of a basic state or a transition, adding more structure or actions. Rules 3 and 4 are mainly used to add a new concern that was not present in the original behavior. These rules add orthogonal components that can be either independent from the remaining components or have an effect on them by broadcasting events. Lastly, Rules 7 and 8 are mainly used to add alternatives to the existing behavior. These rules add new transitions as alternative paths out of a state.

6.3. Practical recommendations for statecharts refinement

From the previous example, and by analyzing the effect of refinement rules on each other, we propose the following rules of thumb to maximize re-use of statecharts using refinements:

1. **Apply the separation of concerns principle to add new behaviors.** As we saw in Section 6.2, statechart refinement allows us to address each concern in isolation, that can be combined later. This in turn reduces the complexity of

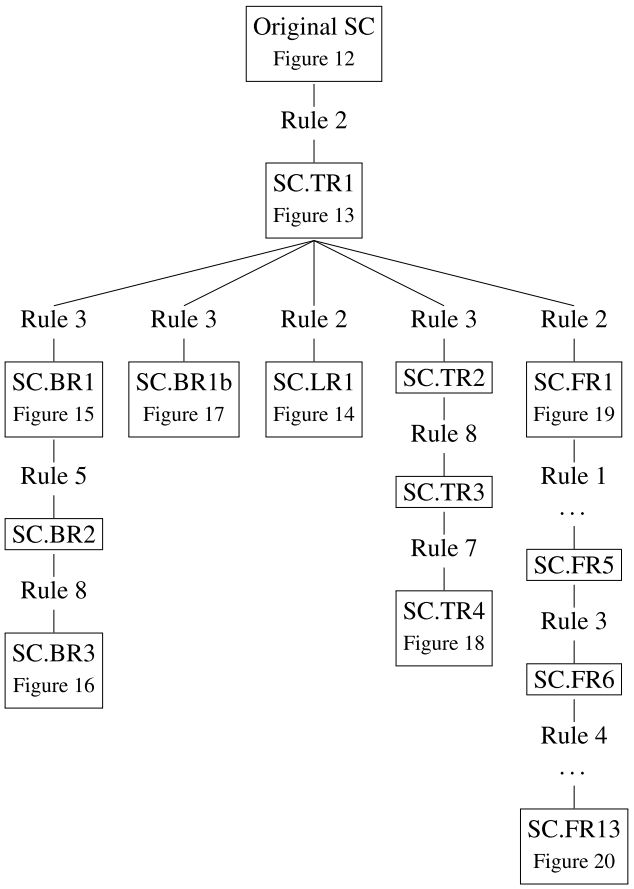

Fig. 21. Refinement evolution tree for the drone example.

dealing with multiple concerns at the same time, making the refinements that deal with a particular concern, easier to create and understand, because they contain only focused information. This is a typical good practice for designing statechart models incrementally. Although this kind of refinement is done in isolation from the remaining statechart, one should nevertheless pay particular attention to possible interactions with other concerns.

2. **Favor refinement of states over refinement of transitions.** Transition refinements are more limited than state refinements or path extensions because of Constraint 3. Furthermore, with our implementation, any guarded intermediate transition must have a dual with the negated guard. Hence, their use should be avoided, if possible, or limited. Many reasons for refining transitions can be formulated as state refinements. Typically, the reason comes from the need to add a series of operations to occur after the event that triggers the transition. An alternative is to refine the target state of the transition, because state refinements have fewer restrictions and offer more flexibility to the user. Another option is to add an orthogonal component that is triggered by the same event. Successive transition refinements become rapidly complex when designing them in a way that preserve the reachability of the target state. Our recommendation is to only refine a transition in order to add a series of actions when the transition is enabled, but before broadcasting an event (if present), or before executing the entry action of the target state. When it becomes too complex, one should consider modifying the statechart directly to already encoding this behavior, thus breaking the refinement chain that guarantees structure and reachability preservation of the original statechart.
3. **Favor events over guards to enable transitions.** This recommendation is a consequence of Constraint 1. The moment we introduce a guard on a transition, we restrict further refinements that can no longer perform actions over the variables used in the guard. A workaround is to follow the strategy of how we obtained SC.FR13 for the drone example in Section 6.1.4. Instead of relying on guards to enable transitions, we use events to trigger them and then add the broadcasting of these events, for example, in separate orthogonal components. It is only then that guards are placed to conditionally broadcast events. This helps in delaying the introduction of guards as late as possible in the refinement evolution tree.
4. **Use of guards to limit further refinements.** This is the reciprocal of the previous recommendation because guards can be used to restrict what can be later refined in the statechart. For example, at some step during the refinement process, we

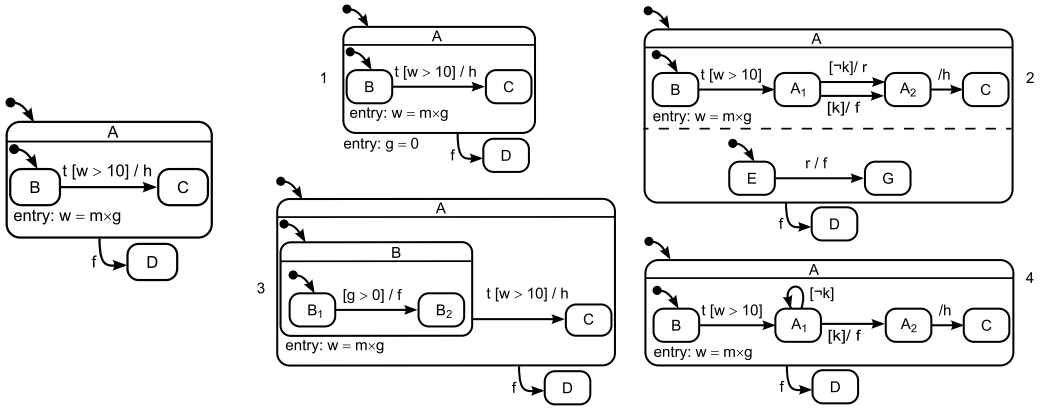

Fig. 22. Set of permissible refinements of (a) that do not preserve the reachability of state C.

may wish to make sure that certain variables cannot be further manipulated to avoid introducing unexpected behaviors. However, one shall make sure that the guard is not too restrictive, since it cannot be later weakened.

A side effect of using our refinements rules is that they tend to add more states than a developer may intend to if he were developing without our rules. For example, revisiting the battery refinement from SC.BR1, we obtain SC.BR3 after applying three rules. This adds two states and three transitions. Instead, a simpler solution could be to simply refine the entry action of `SendWarning` in SC.BR1 to take the decision of when to send the `toLand` event. In practice, there are typically two Statecharts development styles. Some developers tend to minimize the number of states, but add more code in the entry/exit actions, even with conditions or loops. Others try to *model* as much as possible by taking advantage of the structure of Statecharts. The former usually leads to more compact Statecharts, but puts more burden on programmed actions. The latter enables more static analysis by taking advantage of the expressiveness of Statecharts. In that sense, our approach follows the latter style of Statecharts development.

7. Discussion

All our decisions were taken with the intention of making the proposed Statecharts refinement usable in practice and so that the refinement relation can be statically verified between two statecharts. The refinement relation is defined so that the structure of the original statechart is preserved. Given this, we are able to preserve the reachability as much as possible.

7.1. Static verification of refinement rule application

All rules are intended to be statically verified, while constraints specify assumptions on the dynamic behavior of the statechart. This means that, ideally, a dynamic analysis of the statechart, actions, and guards would be necessary. Our implementation of Statecharts refinement presented in Section 5 illustrates a stronger restriction of the application of the rules.

7.1.1. Constraint on actions

In our implementation, we assume that all actions are specified as a sequence of variable assignments. Therefore, Constraint 1 can be statically verified by checking that the variables used in new entry or exit actions of a particular refinement are not part of the set of variables already in use by existing guards. When more complex operations are performed within entry and exit actions, the burden is on the developer to make sure they satisfy Constraint 1. These are the situations where in Section 5.2 the `Validator` can return `WARN`, as a reminder to the developer to make sure of their validity.

Nevertheless, only statically checking if the literal variable name used in a guard is not used in an action, is in practice not sufficient. For example, consider refinement 1 in Fig. 22b. The original guard is on variable `w`, so we cannot perform any further operation over `w`. The new entry action in the refined statechart modifies variable `g`. As `w` depends on `g`, the guard will never evaluate to `true`, and state `C` is no longer reachable with any queue. A symbolic verification of all dependent variables may resolve this issue as done in [3]. However, in practice, this may be very complex when external operations are invoked and the analysis needs to go beyond the scope of the statechart. To mitigate this threat, our implementation reminds the user with a warning message every time an entry or exit action is refined.

7.1.2. Constraint on paths

To check Constraint 4, symbolic analysis of guards is required. Our implementation enforces mutually exclusive guards on transitions having the same event with the same source state (or substate). This restriction makes it possible to verify statically the satisfaction of Constraint 4. It avoids the necessity of determining whether a guard is not a tautology (or contradiction), i.e., given an input queue, there exist two possible configurations such that one makes the guard evaluate to `true` and the other to `false`. Introducing mutually exclusive guards makes sure that the resulting statechart is still deterministic. It also makes sure that the target state of the original path is still reachable. Nevertheless, this still leaves the statechart open to other refinements that break reachability. Take for instance refinement 2 in Fig. 22b where the mutually exclusive guards ensure that A_2 is always reachable, but because both paths generate a sequence of events broadcasting that ultimately trigger transition `f`, `C` is never reached, as imposed by the original statechart. Detecting this situation requires a deep analysis of the possible chains of *steps* that internal events may generate.

7.1.3. Constraint on broadcast

Verifying the satisfaction of Constraint 2 also requires a dynamic analysis of guard evaluation. Our implementation warns the user of this threat when he decides to apply a refinement rule requiring this constraint. However, enforcing a falsifiable guard in new transitions with a broadcast may not be sufficient to ensure reachability. For example, in refinement 3 of Fig. 22b, state `B` is refined such that it broadcasts an event that prevents `C` from being reached.

Constraint 2 cannot be statically verified in all circumstances. The burden of well-formedness is on the user to create guards that make sense in the context of a particular statechart. As a result, this constraint acts only as a guideline for the semantics of the broadcast and guards used, so that if followed, we can have some pre-established assumptions on the behavior of the modeled statechart. Note that our suggested procedure to delay the inclusion of guards helps in maintaining awareness of the semantic context involved, as well as maintaining simpler guards that are more easily manageable to conform to this constraint.

7.1.4. Constraint on transitions

Constraint 3 can have the presence of $\hat{s}_{end}$ states verified, but we cannot statically verify that they are reachable. This occurs due to guard expressions which may not be statically verifiable, like for Constraint 2. Furthermore, the statechart structure itself may contain loops that make the $\hat{s}_{end}$ states unreachable. An example of this is illustrated in refinement 4 of Fig. 22b, where a transition refinement introduces a self-loop. Although this self-loop is guarded, we cannot statically check that this guard is not a tautology, in which case the statechart can reach starvation, making state `C` no longer reachable: the only two possible paths are to remain in state A_1 or to exit state A altogether.

Similarly, we do not support verification of entry and exit functions of basic states to avoid the interpretation of arbitrarily complex action code.

7.1.5. Restrictiveness of our approach

In some regards, the refinement rules we propose may appear too restrictive. With our rules, it is not possible, e.g., to delete an element of the statechart, split states, or modify higher hierarchies by encapsulating existing elements in an OR-state, and refinements must satisfy the four constraints (e.g., preserve existing actions). Nevertheless, the rules guarantee with static verification that the structure and external behavior the original statechart is preserved, leading to safer refinements with respect to regression. Most of the rules allow for a complete statechart model to be introduced in the original, and thus provide extensive freedom to the developer. Furthermore, the restrictions imposed by our approach make it possible to assist developers through proper tooling.

As mentioned in Section 4.2, the set of rules we propose is not meant to be complete. For example, the developer may desire other refinements, such as merging two states into one, splitting a basic state into two, or refinements to additional pseudo-states (e.g., conditionals, junctions). It is possible to add rules for those as long as they preserve structure and reachability. In practice, the eight rules we propose permit most statechart refinements. However, they may require more attention to the order the different aspects of the statechart are approached, following the recommendations in Section 6.3. They may also require in a first instance, less intuitive refinements to reach the desired results. This is reinforced by our evaluation of how to develop the drone example we used to illustrate our approach.

7.2. Refinement vs. modification of Statecharts

In this article we assume a clear difference between modifying and refining Statecharts. Modifying encompasses any operation that changes a Statechart, be it adding new elements, removing them or changing them. In this work, we focus on refinement operations that provide an increment to the Statecharts such that the structure and reachability are preserved. With a stepwise refinement perspective (see Section 6), the developer initiates the statechart development with a more abstract and general behavior to which he keeps adding new information in the form of new Statecharts elements (states, transitions, actions, events, and guards). This assumes the correctness of the original Statechart, so any removal of elements, renaming, rerouting transitions, placing a subset of the statechart inside a composite state, or any other form of modification of the Statechart should be performed on the original Statechart. The refinement approach we propose revolves around the

incremental development of Statecharts. This means that at each new refinement step adds new behavior. We do not consider as refinement any operation that may disrupt the pre-existing behavior.

Nevertheless, to be applicable in practice, the refinements we propose should complement refactoring a statechart model. Assume developers started with a first version of the statechart SC_0 and refined it into SC_n after n refinement steps. At this point, a developer wishes to perform changes that will not preserve the structure, such as refactoring. He proceeds by modifying SC_n leading to SC_{n+1}, with no by-construction static preservation guarantees from any SC_i for $0 \leq i \leq n$. He may be interested in only preserving the behavior of SC_i. Other refinement techniques discussed in Section 8 can then be applied or he should employ other verification techniques (e.g., reusing an existing test suite of SC_i). From then on, he may resume using the refinement rules to preserve properties of SC_{n+1}. In this case, SC_{n+1} becomes the root of a new refinement evolution tree, distinct from the one rooted at SC_0.

7.3. Co-evolution of original and refined Statecharts

Our approach assumes that the base statechart (Fig. 12) is stable and only refinements are allowed to evolve it. In practice, it is often the case that the base design may change. Modifying the base statecharts, as discussed in Section 7.2, follows a different process than our refinement. Issues with synchronization of evolving statechart models have not been dealt with. In our implementation, if the original statechart is modified, then the refined statechart should be discarded and a new refinement process must take place. However, it will be possible to avoid this thanks to the traceability links preserved from the original refinement. A possible implementation can detect if the change affects an element involved in the refinement and use the refinement evolution tree, such as in Fig. 21, to select where to restart the process from.

Certain co-evolutions might also imply changes in the refinements themselves, where the new original statechart impedes the correct application of these refinements. In this case, the refinements must evolve with the original statechart and use the refinement evolution tree to detect when a refinement evolution allows for the effective reuse of some of the previous rules.

In certain model development paradigms (e.g., model-driven engineering), a statechart is often attached to a class definition on which it models its behavior. Refining the class may induce refinement of the statechart that are not supported by our approach, such as non-additive changes. This may also raise further co-evolution issues between the class and the attached statechart.

7.4. Application to other Statecharts variants

There are many variants of Statecharts used in different contexts. Crane et al. [18] compared the syntax and semantics of the three most popular which, together, cover most observed differences. Analyzing their impact on the refinement rules we present in this work, only the case of conflicting transitions requires to adapt the rules to preserve structure and behavior of statecharts. All other aspects that vary between the different Statecharts formalisms are compatible with the presented refinement: they are either syntactic shortcuts that have an equivalent construct in our abstraction of Statecharts (Section 2.1) or the syntax and semantic differences have no effect on the rules.

The only case where our rules are not applicable is when the Statecharts formalism follows the inner-first step semantics to resolve conflicting transitions. As Crane et al. briefly mention, it is possible to mimic outer-first semantics through meticulous use of guards to ensure reachability, but this is not an easy translation that contradicts the choice of such semantics in the first place. The underlying assumption of the stepwise refinement we present is that we want to guarantee that the behavior of the base model must be valid at every refinement increments. All our refinement rules insert a new statechart in a local context to augment the existing behavior, but not contradict it. In that sense, inner-first priority, as in UML State Machines, has the opposite intent by allowing local behavior to override the overall behavior. Therefore such semantics requires a different set of rules to ensure that structure and reachability of the original statechart is preserved.

7.5. Applicability of refinement in practice

There are many practical situations where preserving both the structure and behavior of a statechart is crucial. One such application arises when statecharts are used as implementation artifacts in a software project that requires maintenance. Suppose Alice has developed a first version of the statechart to fit the current requirements. In an agile setting, another developer Bob refines the model to satisfy new requirements. While preserving the behavior of the first version, Bob has significantly changed the structure of the statechart. If now Alice needs to make further improvements on the second version, it will be much harder for her because her expectations of what to look for in the statechart may no longer be met. Structure preservation of Statecharts is crucial for long term development and maintenance: if we preserve the structure and keep the changes made by refinements local (as opposed to arbitrarily changing the structure), we can minimize the effort of future developers. Otherwise, the original design made to guide the development may be lost over time, leading to the same issues present in the maintenance of legacy applications. Furthermore, when an error in the behavior is detected during maintenance, if the structure were preserved at each refinement step, it is possible to trace the error to a specific refinement to be corrected. This facilitates the identification of the source of the problem, but also mitigates the impact of preforming its correction.

Another application where refinements should preserve not only the behavior, but also the structure is when a statechart is refined to include debugging behavior. In this case, we need to guaranty that this augmented behavior does not interfere with the original behavior or structure. In [19], the authors propose to instrument the original model of a real-time system with additional states and transitions to debug the model. The structure of the refined statechart must be faithful to the original, otherwise the developer will not be able to map the instrumented model back to the original model when debugging. The authors define four rules to transform a statechart model into one instrumented with debugging facilities. All four rules can be obtained by composing the application of our refinement rules directly. This illustrates another practical application of our rules, as well as their compositionality as discussed in Section 4.2.

An interesting practical application of our approach is the support the safe construction of product lines of Statecharts. Consider the case where each refinement adds not only new states and/or transitions to a statechart, but also annotates the newly introduced elements with presence conditions. In this case, thanks to both structural and behavioral preservation, it is possible to modularly backtrack to a previous refinement of the statechart in order to make changes, while preserving the product line that was constructed until that refinement.

8. Related work

We compare our work with other approaches to refine different formalizations of Statecharts. We also discuss how other formal approaches garantee behavior preservation of Statecharts and briefly discuss refinement in other behavior modeling formalisms. First, we position our work with regards to the concept of simulation.

In the literature review that follows, we will often mention *simulation*. While the concepts of simulation and refinement are by no means interchangeable and serve different purposes, they do sometimes overlap in practice. For instance, simulation in an operational sense can serve to test if a refinement holds in practice. Also, the formal and idealized notions of simulation, *weak simulation* and *bisimulation* in the context of transition systems [20,21] relate to refinement in the sense that certain types of refinement can sometimes formally adhere to those formal notions. We will also refer to *subtyping* (in the Liskov sense) and *inheritance* as reference frames for our work. In certain circumstances, these concepts do relate, but are in general not equivalent, to our proposal of refinement for Statecharts.

8.1. Refinement and simulation

The refinement relation we introduce in Definition 1 resembles to some extent the widely used definition of *simulation*. According to Definition 1, the identity refinement is a simulation relation (it is in fact a *bisimulation*). However, general refinements following our rules will not necessarily simulate the original statechart. For example, consider a refinement obtained by applying Rule 5 where a transition is replaced by two transitions and an intermediate state. The new state in the refined statechart can perform arbitrary actions (as long as the Constraints 1 and 3 are satisfied), in which case the simulation will no longer hold. This is why we need to introduce the particular notion of refinement in Definition 1. More specifically, we provide in Section 2.3 several examples of how the statechart in Fig. 2 does not simulate the one in Fig. 1, despite being a legal refinement of it.

The refinement relation we have proposed in Section 2.3 is, in some respects, comparable to the formal notion of *weak simulation*. In weak simulation, new behaviors can be added to a system, as long as the observable behavior of the system does not change. In practice, new events occur in the system in a *hidden* form. A refinement step in our approach is comparable to adding a set of hidden events to the statechart, which are processed by the new transitions and states in the refined statechart. The notion of refinement we propose here does not aim at hiding information from the environment, as we rather wish to expose more information resulting from the new behavior.

Barbosa et al. propose in [22] a set of refinement rules that are also aimed at preserving the structure and semantics of Statecharts. The operational Statecharts semantics they consider is introduced via extended hierarchical automata [23], for which semantics are given in an algebraic manner. The notion of bisimulation is then used to decide on Statecharts equivalence. They introduce a set of simple refinement laws that are behavior preserving, which can be composed to form more complex refinement patterns, like ours. Removing parts of the statechart is allowed by these laws, contrarily to our approach, however an alternative to the removed behavior must be present. Similarly to us, Barbosa et al. provide additional constraints that need to be verified in order for the laws to yield sound refinements. However, the authors do not provide formal proofs that the allowed refinements are sound.

8.2. Refinement approaches of Statecharts

Several works in the literature have explored the concept of Statecharts refinement. Our work on refinement focuses on the preservation of pre-defined structure and reachability properties in the process of defining new details in the original model. Most of the works looked at the problem in the context of class inheritance, where a subclass can inherit from a superclass' statechart. We therefore compare how this Statecharts specialization is performed in contrast with the Statecharts refinement we propose. Table 1 summarizes the comparison of different refinement approaches present in the literature with ours. Like in this work, most approaches propose specific refinement rules we briefly describe:

Table 1
Comparison of Statecharts refinement approaches along different rules and features.

Refinement approach Variant applied on	**Ours** **Statemate**	[7] Stateflow	[4,25] μ-Charts	[24] Flat state machines	[8] Flat state machines	[26] Flat state machines	[3] UML-B	[9] B Method
Element removal	**N**	N	R	Y	_	Y	_	_
Guarantee regression	**Y**	R	R	Y	R	N	R	R
Add states	**R**	Y	R	Y	Y	Y	Y	Y
Convert to complex	**Y**	Y	Y	n/a	n/a	n/a	Y	Y
Add orthogonal	**Y**	Y	Y	n/a	n/a	n/a	Y	Y
Preserve nesting	**Y**	N	N	n/a	n/a	n/a	_	Y
Preserve transitions	**Y**	_	_	_	Y	_	_	_
Split state	**N**	_	_	Y	_	_	_	_
Add actions	**Y**	_	_	_	Y	Y	_	_
Preserve actions	**Y**	_	_	_	Y	N	_	_
Add variables	**Y**	R	Y	_	Y	Y	Y	Y
Refine pseudo-states	**N**	_	_	_	_	_	_	_
Add transitions	**R**	Y	R	Y	R	Y	Y	Y
Preserve source	**Y**	Y	_	_	_	_	_	_
Preserve target	**Y**	_	_	_	_	_	_	_
Preserve trigger	**Y**	Y	R	_	_	_	_	_
Preserve guards	**Y**	Y	R	_	_	_	_	Y
Preserve broadcasts	**Y**	Y	_	_	_	_	_	_
Split transition	**R**	_	_	_	_	_	R	_

Legend
Y fully supported
N explicitly not supported
R restricted support under specific conditions
_ not defined
n/a not applicable

- Variant applied on: the Statecharts formalism variant the refinement approach is applied on in the referenced paper.
- General rules:
  - Element removal: removing states and transitions without restriction.
  - Guarantee regression: presence of any mechanisms to verify the preservation of behavior.
- State refinement:
  - Add states: adding new (basic) states to a statechart without restriction or by connecting it to existing elements.
  - Convert to complex: refining a state into a more complex state, i.e., Basic-to-OR, OR-to-AND and Basic-to-AND.
  - Add orthogonal: adding new orthogonal components, i.e., AND-to-AND.
  - Preserve nesting: preserving the relation with a parent state when refining a state.
  - Preserve transitions: preserving all incoming and outgoing transitions when refining a state.
  - Split state: splitting a state and its components into multiple states.
  - Add actions: adding new entry and exit actions in a state.
  - Preserve actions: preservation of pre-existing entry and exit actions of a state.
  - Add variables: adding and using new variables without any restriction.
  - Refine pseudo-states: presence of rules specific to pseudo states, except their addition or removal.
- Transition refinement:
  - Add transitions: adding transitions between two arbitrary states without any restrictions.
  - Preserve source: preserving the source state when refining a transition.
  - Preserve target: preserving the target state when refining a transition.
  - Preserve trigger: preserving events that trigger an existing transition.
  - Preserve guards: preserving guards on existing transitions.
  - Preserve broadcasts: preserving broadcasting events on existing transitions.
  - Split transition: refining a transition into a path composed of multiple transitions and at least one intermediate state.

All the refinement approaches allow removing an element of the original model, except for refinements in Stateflow and in ours. Our approach and is the only one that provides static regression guaranties over the behavior. The approach in [24] only preserves behavioral regression on flat state machines. Furthermore, approaches based on μ-calculus, B, and as described in [7] require additional proofs or simulations. The remaining approaches do not integrate any such guaranties or analysis in the refinements. In the following, we provide further insight than reported in Table 1 on the particularities of each approach.

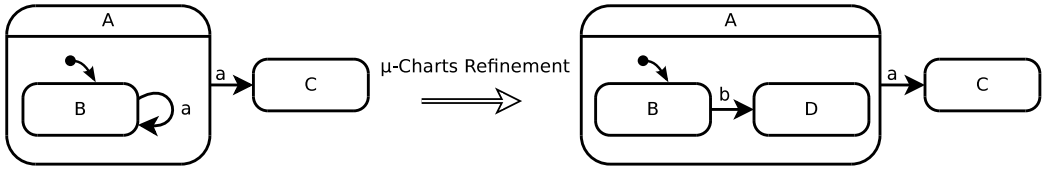

Fig. 23. An example of a μ-Charts refinement that is not valid by our rules.

8.2.1. Simulink/stateflow

Simulink [27] uses a causal block diagram notation to model control systems. It provides blocks, which are similar to states in Statecharts, and ports, which are the input/output of the blocks. Ports are capable of having preconditions or postconditions defined on them. Stateflow charts [28], a variant of Harel's Statecharts, are introduced within Simulink models to explicitly model the behavior of the reactive system. In what concerns refinements for Stateflow charts only, most of the literature focuses on transforming Stateflow into formal languages that support refinement, such as in [29]. Nevertheless, Boström et al. [7] presented a refinement calculus for Simulink to provide additional capabilities for stepwise development, which is not included by default within Simulink. As with our approach, the purpose of the work from Boström et al. is to make common development practices more predictable and usable. Their refinements focus on two main goals: to create an implementation of a specification and to add features to an implementation, otherwise referred to as superposition refinement.

Specifications are similar to abstract classes, in that they simply define the overall structure for an implementation to follow. These specifications do not contain actual implementation details within them. During the refinement of a specification into a concrete implementation, several criteria must be met, summarized in Table 1. Superposition refinement aims at adding new behavior to a concrete model while preserving the original behavior of the model. New unconnected ports may be added to the refined model, however these ports cannot have preconditions attached to them. Concrete blocks may only be added and cannot be removed from the original model. This is similar to our requirement that states or transitions shall not be removed during refinement. Any block within the model that are marked virtual can be freely added or removed during refinement. We do not have this concept and do not allow for any states or transitions to be removed, as in our solution they may contain needed implementation details already.

8.2.2. Refinement calculus for μ-Charts

The μ-Charts formalism [4,25] was introduced as a variant of Statecharts that sought to provide additional expressive power. As with our solution, μ-Charts are based on Harel's Statemate semantics of Statecharts. In these papers, Scholz defines a calculus for refining μ-Charts as part of an incremental design process.

Transitions in μ-Charts can be added as long as the event trigger and guard associated with the new transition do not conflict with other transitions. In particular, the calculus requires that in composite states, any outgoing transition of the new substates does not broadcast events that are already being broadcast within that composite state (i.e., broadcasts must be unique). This is somewhat similar to our Constraint 4. We do not make this stipulation, as there is no need to limit broadcasts to this extent to preserve structure and reachability. Instead, we require that the transitions with these new broadcasts be guarded.

Transitions can be removed only if there is another transition present that is enabled with the same event and guard evaluation as the deleted transition, and if the latter is never able to be triggered. Guards can be added through either conjunction or disjunction, as long as several formal conditions hold. Events and guards can also be removed if certain formal conditions hold. We do not allow for events or guards to be modified in our solution due to our focus on preserving the external behavior of the Statechart. Fig. 23 shows an example where an internal transition with event a has been removed, since event a will always trigger the outer transition first. The figure also illustrates a refinement where an additional transition and path have been added out of state B. This would be valid with Rule 7 in our rules if transition a was preserved.

The μ-Charts rules allow removing default states, which is equivalent to removing a sub-statechart. Additional states can also be added, as long as they are plugged into the existing flow using new transitions. This is similar to Rules 5–8, as we require new states to be connected with new transitions or part of a valid statechart that is a component of a connected state.

8.2.3. Refinement on flat state machines

The authors in [24] define State Transition Diagrams with a semantics based on Statecharts, but where the focus is on asynchronous communicating systems such as in I/O-Automata. They present a refinement process where only the I/O interface of the machine is preserved. In contrast with our approach, this ignores variables and actions that may occur inside the states which allows us to guarantee reachability preservation statically. Furthermore, the authors define rules for addition and removal of states and transitions for flat state machines which would breach structure preservation in our approach.

In [8], the refinement is defined as an inclusion of the input/output trace between the refined and the original statechart. This is a black-box approach where internal events and the structure of the statechart are ignored. Since they allow non-determinism, refinements that remove states and transitions are defined to resolve the presence of chaotic behavior. Our approach does not treat the statechart as a black-box, and focuses instead on enforcing backward compatibility of the behavior, and the structure and internal events of the statechart.

Paech and Rumpe have introduced in [26] a type-centered technique where they equate automata with types. Specializing a type by adding attributes or operations to it is then reflected in the equivalent automaton in the form of adding data or transitions to it. The authors shown that such type specializations do preserve the fact that, regarding an environment (also defined using types), the supertype and the specialized subtype cannot be distinguished. Consequently, this implies a refinement relation between the corresponding automata. Besides presenting rules that do preserve the specialization/refinement relation, the authors also prove that such a relation is transitive, as we do in our work.

8.2.4. Refinement in UML-B

In [3], Event-B is used to refine UML class diagrams, and their semantic description defined in a state machine. This is done by translating the state machines into a state set representation or a state function representation. The refinement of the state machine is achieved in one of two ways. One is the refinement of transitions into multiple parallel transitions that specify subcases of the original transition. The other is the elaboration of a state with a nested state machine. This is similar to Rule 2, but without giving a concrete resolution to the situations where the state being refined already has a nested state machine, though not explicitly disallowing it. Most of the presented rules that guide these two refinement patterns deal with the preservation of the elements unaffected by the refinement, while we make use of a notion of identity, that helps us isolate this preservation of elements. While we aim at proving static verification of preservation of the refined statechart, the work in [3] executes an evaluation using theorem provers that, in conjunction with gluing expressions, checks the preservation of the original reachability of the state machine. Gluing expressions are used to provide the translation of the original conditions in terms of the refined elements. These can be manually defined or the evaluator can try and complete them using a discoverer, in accordance to the defined parameters and the free variables present. This leaves the possibility, although unlikely, of having gluing expressions of successive refinements that lead to a complete loss of the original semantics.

With the refinement of transitions, although different mechanisms of refinement are used, both provide similar results. In UML-B, the transitions are refined into subcases of the same transition as parallel transitions, while in our work we split the transition into a sequence of transitions where the subcases can be addressed between intermediate states of this path. This makes our approach more susceptible to the order of refinements, but with harder guaranties of preservation of the original conditions to enable the transition.

Another situation that is not approached in UML-B state machine refinement is the interference of orthogonal components and other sections of the statechart that is covered by our guards. Our approach is more restrictive, but covers more situations with respect to Statecharts semantics, while the approach presented for UML-B refinement allows for greater flexibility and evaluates guard expressions themselves, at the cost of having to prove reachability for every refinement.

8.2.5. B method

The well-known B-method [9] proposes using the Abstract Machine Notation (AMN) to represent specifications throughout the software development process. A mathematical theory to refine AMN specifications allows one to add the necessary detail from requirements through implementation, while formally guaranteeing that relevant invariant properties of the system always hold. Each AMN machine contains a set of variables and a set of operations for manipulating those values and communicating with the environment. Although it also describes behavior, a B machine is different from a statechart in the sense that states are not explicitly modeled. Nevertheless, there are several resemblances between the notion of B refinement and our proposal of statechart refinement.

As in our approach, refinements in B aim at preserving the structure and reachability of a machine. Also, the behavior of a refined B machine can be more detailed than that of the original machine, e.g., by eliminating non-determinism. As in our approach, sequences of events in the traces generated by the B machine can be interleaved by additional detail in the sequences of events in the traces of a refined B machine. Furthermore, the structural preservation condition in Definition 1 resembles the hiding principle of import rules in B refinement. The import rules ensure that the structure of the specification and the system's invariants are preserved throughout refinement. Finally, as in our case, the refinement relation in the B-method is also proved to be transitive.

There are also important differences between the two approaches. Firstly, refinements in B can be done over variables and/or the internal specification of operations. Our refinement relation concentrates on the notion of state, while abstracting from the data types manipulated by the entry or exit operations inside states. Our approach does nonetheless enforce certain syntactic constraints on the guards of the refined statechart transitions, thus implying limitations on the data flow possibilities through the refined statechart. Secondly, proof obligations are necessary in B to formally show and document that the invariants that were originally stated about a machine still hold. Proof obligations are generated on-the-fly for a given B refinement. This allows refinements of B machines to be very flexible, as long as the operations affected by the invariants remain structurally the same. However, the burden is on the developer to demonstrate that invariants hold manually or with automatic assistance. In our case, our refinements are less general and allow less flexibility because the

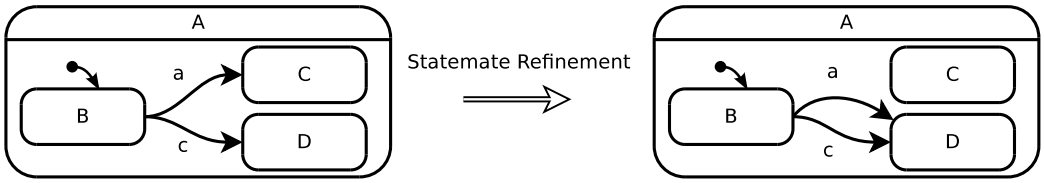

Fig. 24. An example of a Statemate refinement that is not valid by our rules.

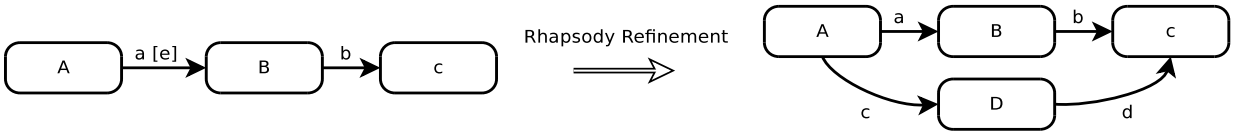

Fig. 25. An example of a Rhapsody inheritance refinement that is not valid in our rules.

rules are predefined and do not concern data types in general. Furthermore, our approach guarantees the preservation of structure and reachability of the original statechart, caveat the limitations discussed in Section 7 because of the tradeoff between theory and practice.

8.3. Refinement for subtyping

Subtyping is related to the notion of refinement, yet it serves a different purpose. We analyze some Statecharts subtyping approaches for additional incites on Statecharts refinement.

8.3.1. Statemate

Harel et al. [2] address the concept of Statechart refinement through the use of object oriented inheritance, laying out several basic rules for states and transitions. As with our implementation, the focus for this approach is on creating functional and pragmatic rules to govern the refinement operations that can be performed on a statechart. To add new states, the developer can apply any of our rules (except Rule 1), but connectivity is mandatory to make sure the new states are reachable.

Harel et al. also provides several rules for transitions refinement. The target state and guard of a transition can be changed as desired. Such refinements can easily break the original structure, as demonstrated in Fig. 24. By having stricter structural rules, we can provide a more consistent and predictable relationship between the original and the refined statecharts.

8.3.2. Rational rhapsody

Rhapsody [30] defines three types of modifiers that can be annotated on a statechart: *inherited*, *overridden*, and *regular*. Elements marked as being inherited will allow for changes to the original statechart to propagate to the refined model. Elements marked as being overridden no longer accept general changes from the original statechart. However, deletions from the parent will still propagate to the child. Overridden statecharts have no relationship to the original statechart. In our approach, we do not provide the concept of overriding the original model. If the original structure and reachability is no longer desired, then the original-refined statecharts relation no longer holds and the new statechart should be developed independently from the original, as discussed in Section 7.3. If a new behavior is desired in the refined model, then that behavior must be added following the refinement rules.

There are several Statecharts inheritance rules in Rhapsody, but are less restrictive than ours at the cost of not preserving neither the structure nor the behavior. For example, Fig. 25 shows a valid refinement in Rhapsody, where state D and transitions c and d have been added. In contrast, in our approach, the removal of the existing guard is not permitted, but the new path can be created using Rule 8 with the added restriction that the initial transition c is deterministic and does not interfere with a applies.

8.3.3. UML state machines

UML presents three sets of inheritance rules for State Machines [31]: *subtyping*, *strict inheritance*, and *general refinement*. We only focus on the former two, because the latter allows modifying the statechart almost arbitrarily.

Subtyping As in our work, the focus of subtyping is to preserve the pre and postconditions of applying the events and actions of a state machine, thereby satisfying LSP. When refining with subtyping in UML, a refined transition must remain connected to its original source state, though it can have its target state altered. We do not allow for altering the source or the target state of transitions, as this would break the semantics of the original state machine. Guards are allowed to utilize conjunctions, which may invalidate the enabling of a transition that would have been enabled in the original. The

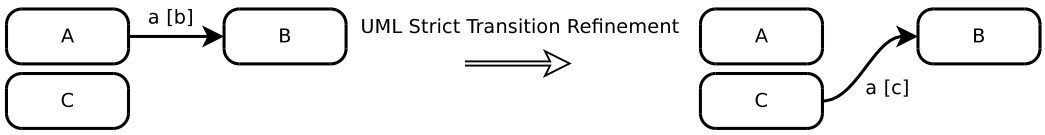

Fig. 26. An example of UML transition strict inheritance that is not valid in our rules.

specification in [31] argues that the states preserve the output of the statechart as a whole, while the transitions define how each state is related to other states. In our rules Condition 3 prevents altering existing guards from being changed. This decision was made so that we can only restrict the state space expected by any refined statechart, leaving us within the originally defined state space. In UML subtyping, it is possible to effectively remove the effect of a guard by disjuncting it with `true`.

Strict inheritance The focus of strict inheritance is to re-use a statechart associated with a parent class [31]. The rules are very permissive which hinders regression guarantees. Fig. 26 demonstrates a transition following the strict inheritance rules of UML that violates the preservation of the identity of the refinement definition as the path from A to B no longer exists. In our rules, we only permit the creation of a new path between C and B if there already exists a way of reaching B from C.

Strict inheritance and subtyping provide greater freedom than permitted by our rules. Our focus is to provide more predictability and clearly defined expectations of any refined statechart than the rules provided by UML.

8.4. Refinement in other behavior modeling formalisms

8.4.1. Interface automata

Interface Automata [32] are an alternative formalism to Statecharts for designing reactive systems. This formalism utilizes, at its base, a component structure which exposes a set of expected inputs and outputs. Components can then contain a set of states and transitions which fire on specific inputs. It is assumed that the executing environment treats each component as black box.

Refinement of Interface Automata is concerned with following the principle of contravariance. This is closely related to our intention of preserving the external behavior of a Statechart during refinement. These refinements have complete freedom in terms of the creation, removal, or modification of existing states and transitions, so long as the refinement respects the state space of the input and output events, since this solution seeks to preserve aspects of the input and output events of the component being refined. We do not grant this freedom in our solution since we do not assume that components are black boxes from the environment.

8.4.2. Partial behavioral models with uncertainty

The work by Chechik et al. [33] on partial modeling with uncertainty looks at behavioral models, such as Statecharts, and their refinements from three points of view. Typically, a statechart describes how the modeled system shall behave (positive set) and, consequently, how it should not behave, often defined by the complement of the traces output by all its possible execution (negative set). However, the approach by Chechik et al. allows the modeler to explicitly specify what parts of the positive or negative sets are deemed uncertain at earlier stages of the development. To achieve that, they annotate the elements of a model with three possible values: *true*, *false*, and *maybe*. The refinement of a state machine will reduce the uncertainty in the partial model, thereby replacing *maybe*'s by *true*'s and *false*'s. Unlike their approach where the modeler can freely modify a *maybe* element, we only allow for a limited set of modifications to refine the model. The advantage of our approach is that refinements are guaranteed to preserve the structure and reachability of the original model, while their approach requires to explicitly verify these properties as in the B-method.

8.4.3. Petri nets

Petri nets are another formalism for modeling the behavior of systems. Since Petri net models are also developed incrementally, there is a lot of literature on guaranteeing the preservation of certain properties when refining models [34–37]. In previous work [38] we have explored the conditions under which refinements of a Petri net model preserves invariant safety properties by construction. Place preservation and strengthening of guards on transitions are similar to our rules for Statecharts refinement.

8.4.4. Specification and description language

The Specification and Description Language (SDL) [39] is a hierarchical modeling language that is an extension of finite state machines. It defines reactive systems similar to those represented by Statecharts. Systems are the root elements, which contain blocks. Blocks contain processes. Processes contain procedures. The procedures contain the operational statements in the model, while the other layers contain different levels of abstraction for signal routing. The SDL specification allows for specialization/inheritance, however there is no specific definition for an SDL equivalent of our refinement rules.

SDL specialization results in a type-subtype relationship between original and specialized model elements. SDL elements can be marked as virtual, which allows for the element to be completely redefined. Elements that are not marked as virtual must be at least partially preserved. All properties must be preserved during specialization. Transitions can be redefined during specialization. Additionally, a subtype should only be expected to have one concrete super-type, though the specification allows for multiple interfaces to be implemented. Interfaces define signals, signal lists, or remote procedures that the implementation will require. This is similar to languages such as Java which allow for multiple interfaces to be implemented but only one class to be extended. This implementation is overall very similar to the inheritance implementation in Rhapsody.

8.4.5. Abstraction patterns

In [40], the author presents a series of structural, behavior and temporal abstraction patterns, with the aim of better help in reducing the discrepancies between design aim and actual implementation. Therefore these patterns are related to our refinement rules when applying their inverse. Most of the patterns presented in the structural and behavioral categories rely on the composition or a variation of the Black Box and Cable patterns. In particular, the Summary State and the Summary Message patterns map closely to our Rule 2 and 8 respectively. However, to be applicable to Statecharts, the inverse of the Summary State pattern would require additional information to not break the semantics and structure of the original Statechart. Apart from the Group Transition pattern that allows one to fundamentally change the structure, all other patterns can be achieved by composing our rules. This reinforces the breadth of applicability of the possible refinements achievable with our approach.

8.4.6. Software behavior refinement

The FOCUS software development method from Broy and Fuchs [41] provides a thorough description of how distributed software can be developed in a component-based formal fashion. FOCUS encompasses the complete software development cycle, from requirements to code. In FOCUS, software development is achieved incrementally, via many different types of behavioral or structural refinements, adapted to the current model and methodological step in the development cycle. While FOCUS is a collection of mathematical models, concepts and rules, its tool counterpart AutoFOCUS [42] allows building hierarchical state machines as models of the behavior of individual components. The FOCUS theory provides a formal background for the integrated semantics of all types of models proposed in AutoFOCUS and also a set of refinement rules to be applied during the software lifecycle. Broy and Fuchs require that formal proofs following refinements are built, such it can be shown that the development steps are theoretically sound. In the AutoFOCUS tool these principles are loosely applied and, while soft principles of model correctness are applied to the models that are built (e.g., conformance to a metamodel), a gap with the theory remains to allow for usability in practice.

9. Conclusion

In this paper, we presented a new refinement relation to support the control of Statecharts refinements. Unlike other approaches, we restrict refinements to guarantee the preservation of the structure and reachability of the original model. We presented a set of refinement rules that are consistent with these constraints, while taking into account certain practical and real-world considerations. We formally demonstrated the soundness of these rules and proved the reflexivity and transitivity of the refinement relation, thus allowing for incremental, multi-step refinements. We implemented and tested our refinement rules on an example of incremental behavior modeling to demonstrate how the original structure and reachability is preserved, while also granting extensive development freedom. We foresee that the application of this refinement relation is not only useful for the re-use of statechart models, but can also serve to better design statechart models, following a stepwise refinement development strategy.

Our approach relies on striking a balance between practice and theoretical guarantees: our method preserves theoretical guarantees as much as possible, while retaining the possibility of performing reasonable refinements by using a set of fixed rules. While in some cases it is possible to fully achieve formal guarantees of reachability using the rules we propose, in other cases additional constraints (formally identified in this work) also require satisfaction. In these cases, additional methods and/or tools need to be used to ensure the satisfaction of such constraints. This is fully compatible with the way in which software is developed in practice: specifications are initially described in the form of requirements, designs, and high-level properties, but, in most cases, software development is an attempt to approximate them. During the verification phase, testing, simulation or, in some punctual cases, model checking or theorem proving is used in order to increase confidence that the original specifications are met. We propose a method that guarantees preservation of behavior up to some point, while leaving the remaining cases to be covered by the modeler using additional verification methods that we leave, for the time being, open. We believe this goes a step further than existing methods for statechart refinement: we propose a formal framework that provides a solid reference for a notion of statechart refinement while compromising with practice in a well-specified and formal manner.

From the implementation point of view, our current prototype allows the modeler to start from the original statechart model, freely add states and transitions to it, and then verify that it is a valid refinement. An alternative implementation may offer more automation using a model transformations encoding the refinement rules. However, this transformation can only be applied interactively since we cannot predict how many new elements the modeler desires to introduce. In the current

implementation, the burden is on the developer to satisfy the constraints on the behavior of the refined statechart. We would like to provide better tool support to reduce this effort. We also plan to investigate how to generalize our refinement framework to accommodate other Statecharts semantics, since we currently only consider the usage of Statemate semantics. Finally, although most decisions were taken with practicality in mind, we plan to conduct more studies to determine edge cases associated with our approach.

Appendix A. Proofs of Section 4

A.1. *Lemma 1: Local rule application is a (local) refinement*

Refinement rules create (local) refinements restricted to the elements of the statechart being refined and satisfy the five refinement conditions in Definition 1. Formally, given a statechart SC, and a local rule application (SC, R, SC_r), we have that $SC_{lo} \overset{R}{\preceq} SC_{lr} \in \Re$ where: SC_{lo} is the statechart that defines the local scope of application of the rule, as stated in Section 2; and SC_{lr} is the statechart composed of only the states and transitions of SC_r that are mapped by R onto SC.

Proof. Condition (1): From Definition 2, regardless of the refinement rule used, every state or transition in SC_{lr} is mapped to exactly one state or transition in SC_{lo}, hence R^{-1} is surjective.

Condition (2): Trivially true for *Basic-state to Basic-state* and *Basic-state to OR-state* refinement rules, because no state hierarchy exists in the state s being refined. When an *OR-state to AND-state* refinement rule is applied to a state s (Rule 3), for any state p such that $(p, s) \in in_l$ we have that $(p_r, s_r) \in in_{lr}^*$, where $(p, p_r), (s, s_r) \in R$ given p remains indirectly attached to s via the newly created AND-state. When an *AND-state to AND-state*, *transition*, *state extention* or *path* refinement rule is applied (Rules 4, 5, 7 and 8), all the state hierarchy is preserved given that by Rule 4 $in_l \subseteq in_{lr}$ and that by Definition 2 we have $(s, s) \in R$ for any $s \in S$ and $(t, t) \in R$ for any $t \in T$.

Condition (3) is satisfied by design since, regardless of the refinement rule used, new variables can be introduced in the entry/exit actions of the new states in S_{lr} and new events can be introduced on the new transitions in T_{lr}. All refinement rules Rules 1–8 guarantee this condition by construction.

Condition (4): Trivially true for any state refinement rule (Rules 1–4, 7 and 8), because the transitions going inside and outside of the state being refined are not touched by the refinement, and in particular in Rules 7–8, by construction any added transition does not interfere with pre-existing transitions. For the transition refinement rule (Rules 5–6), the source and target states of a refined transition t are preserved as well as all transitions adjacent to them. A new transition is built from the source state of t into the initial state of the statechart used as a parameter for the refinement. Another new transition is built from the parameter statechart end states into the target state of t, thus satisfying the condition.

Condition (5): Reachability preservation is guaranteed in three ways. First, since we only modify new variables, any state reachability that depends on $en(s)$ and $ex(s)$ is preserved in $en_r(s)$ and of $ex_r(s)$. This means that we have $\forall v \in V \cap V_r, g(\xi(v)) \rightarrow g_r(\xi_r(v))$ and any action on variables in $V_r \setminus V$ does not impact reachability. Second, through Constraint 2, we only allow a new event broadcast if there exists a queue where its transition is not enabled. These new broadcasts could trigger events in such an order that would render some states of the statechart inaccessible. By only allowing these new broadcasts with the presence of guards, we establish a range of executions where the new broadcast is not triggered, resulting in the preservation of the original reachability conditions. Third, with Constraint 3, we establish that the application of Rules 5–6 will not introduce dead-ends where previously there was a path. □

A.2. *Property 1: Refinement reflexivity*

The identity is a refinement $SC \overset{R_{id}}{\preceq} SC \in \Re$.

Proof. To demonstrate the reflexivity of $\Re$, we need to show that each refinement condition allows for the identity and consequently from Lemma 1, each refinement rule allows for the identity.

Condition (1): Since the identity mapping is a bijection, then R^{-1} is surjective because $id^{-1} = id$.

Condition (2): Holds because $(s, s') \in in$ implies $(s, s') \in in_r$ and $(s, s), (s', s') \in R$.

Condition (3): Satisfied because $\subseteq$ is reflexive.

Condition (4): We have that $\forall (s, s), (s', s') \in R$, if $s \xrightarrow{e[p]/x} s' \in T$ then $s \xrightarrow{e[p]/x} s' \in T_r$, and thus all connections in the statechart are maintained.

Condition (5): By definition of the identity, all transition guards, events and broadcast are the same for any t and t_r where $(t, t_r) \in R$. Also, all entry and exit actions of states are the same for any s and s_r where $(s, s_r) \in R$. Therefore, if a transition t is enabled in SC, its corresponding refined transition t_r will be enabled in SC_r. Hence if a state s is reachable in SC, its corresponding refined state s_r will be enabled in SC_r. □

A.3. *Lemma 2: Rule application is a refinement*

The application of a *state refinement* of a *transition refinement* rule is a refinement $merge(id, loc) = SC \overset{R}{\preceq} SC_r \in \Re$ where $id = SC \overset{R_{id}}{\preceq} SC$ and $loc = SC \overset{R_{loc}}{\preceq} SC_r$.

Proof. We can always separate the merge of R_{id} with a local refinement R_{loc} in three parts: $R_1 = R_{id} \setminus R_{loc}$, $R_2 = R_{loc} \setminus R_{id}$ and $R_3 = R_{id} \cap R_{loc}$.

Condition (1): $R_1 \cup R_3$ is inversely surjective because it is a subset of R_{id}. From Lemma 1, we have that R_{loc}^{-1} is surjective. Furthermore, note that the domain of R_{loc}^{-1} has no common elements with the domain of R_{id}^{-1}. Therefore R_1, R_2, and R_3 are each inversely surjective, hence $merge(id, loc) = R_1 \cup R_2 \cup R_3$ is inversely surjective.

Condition (2): Because $R_1 \cup R_3$ are a subset of R_{id}, and by definition of identity all containment relations are kept unchanged. By construction Rule 3 ensures that the altered containments relation maintain a path to the originally contained state, and all remaining Rules are only able to add to the hierarchy, so R_2 preserves its hierarchy. Because hierarchy relations are unique the $merge(id, loc) = R_1 \cup R_2 \cup R_3$ maintains the hierarchies unchanged.

Condition (3): Like before, because $R_1 \cup R_3$ is a subset of R_{id}, they keep all events and variables unaltered. Because the refinement rules can not alter or remove events and variables, only create new ones, we have that R_2 also preserves this condition. Because we can only add events and variables, the $merge(id, loc) = R_1 \cup R_2 \cup R_3$, will also only add events and variables, making it still preserve the condition.

Condition (4): From Lemma 1, we have that R_{loc} is a refinement, so $R_2 \cup R_3$ conforms to this condition. By definition, R_{id} also preserves the connectivity between states. Because the connectivity in $R_1 \cup R_3$ and in $R_2 \cup R_3$ and there is no path that connects between states of R_1 and R_2 without passing through R_3, the connectivity of $merge(id, loc)$ is also preserved.

Condition (5): Similarly, by definition of identity, $R_1 \cup R_3$ maintains its reachability, and as shown in Lemma 1, $R_2 \cup R_3$ also holds this condition, guarantying that it cannot introduce new actions that affect pre-existing variables, including variables that exist in $R_1 \cup R_3$. Because pre-existing action cannot affect new Variables and we guard against new actions affecting pre-existing variables, $merge(id, loc)$ will preserve the reachability of all its states. □

A.4. *Property 2: Refinement transitivity*

If $SC \overset{R_1}{\preceq} SC_{r_1} \in \Re$ and $SC_{r_1} \overset{R_2}{\preceq} SC_{r_2} \in \Re$, then there exists a refinement $SC \overset{R_3}{\preceq} SC_{r_2} \in \Re$.

Proof. To demonstrate the transitivity of $\Re$, we need to show that each refinement condition is transitive.

Condition (1): Since R_1^{-1} and R_2^{-1} are surjective, then $R_3^{-1} = R_1^{-1} \circ R_2^{-1}$ is surjective.

Condition (2) is satisfied because *in* is transitive.

Condition (3): Satisfied because $\subseteq$ is transitive.

Condition (4): Suppose we have $s \xrightarrow{e[p]/x} s' \in SC$. If we name the resulting connectivity results presented in Condition (4) as (1) $\exists s_r \xrightarrow{e[g]/x} s'_r \in SC_r$; (2) $\exists s_r \xrightarrow{e[g]} s''_r \xrightarrow{\varphi[true]/x} s'_r \in SC_r$; and (3) $\exists s_r \xrightarrow{e[g]} s''_r \xrightarrow{+} s'''_r \xrightarrow{\varphi[true]/x} s'_r \in SC_r^+$. The application of $SC \overset{R_1}{\preceq} SC_{r_1}$ will produce (1), (2) or (3). We can apply R_2 to any of these results, where if R_1 resulted in (1) the application of $SC_{r_1} \overset{R_2}{\preceq} SC_{r_2}$ to it will produce results of type (1), (2) or (3). If R_2 resulted in (2) the application of R_2 will result in (2) or (3). And if R_1 results in (3), R_2 will result in (3). In any of these cases they remain within the results (1), (2) and (3) that preserve the connectivity of the Statechart.

Condition (5): Knowing that R_1 and R_2 are refinements. If we assume that $R_3 = R_1 \circ R_2$ is not a refinement, implying that $\nexists Q_{r_2}$ that allows $I_{r_2}, Q_{r_2} \xrightarrow{bigstep} (\bar{S}_{r_2}, \xi_{r_2}, \eta_{r_2})$. Then as R_3 produces the same refinement as $R_1 \circ R_2$ this means that either the application of R_1 or R_2 is preventing the existence of Q_{r_2}, implying that one of them is not a refinement, leading to a contradiction. Therefore there must be at least one Q_{r_2}, proving that R_3 must be refinement. □

A.5. *Property 3: Sequence of rule applications is a refinement*

$\forall n \geq 3$, the composition of rule applications $SC_1 \overset{R_1}{\preceq} SC_2 \overset{R_2}{\preceq} \cdots \overset{R_{n-1}}{\preceq} SC_n$ is a refinement.

Proof. The proof follows from Lemma 2 and Property 2. □

References

[1] D. Harel, Statecharts: a visual formalism for complex systems, Sci. Comput. Program. 8 (1987) 231–274.
[2] D. Harel, E. Gery, Executable object modeling with statecharts, Computer 30 (1997) 31–42.

[3] M.Y. Said, M. Butler, C. Snook, A method of refinement in UML-B, Softw. Syst. Model. 14 (2015) 1557–1580.
[4] P. Scholz, Incremental design of statechart specifications, Sci. Comput. Program. 40 (2001) 119–145.
[5] D. Harel, O. Kupferman, On object systems and behavioral inheritance, IEEE Trans. Softw. Eng. 28 (2002) 889–903.
[6] N. Szasz, P. Vilanova, Behavioral Refinements of UML-Statecharts, Technical Report RT 10-13, Universidad de la República Montevideo, 2010.
[7] P. Boström, L. Morel, M. Waldén, Stepwise development of simulink models using the refinement calculus framework, in: Theoretical Aspects of Computing, in: LNCS, vol. 4711, Springer, 2007, pp. 79–93.
[8] C. Prehofer, Behavioral refinement and compatibility of statechart extensions, Electron. Notes Theor. Comput. Sci. 295 (2013) 65–78.
[9] J.-R. Abrial, M.K.O. Lee, D.S. Neilson, P.N. Scharbach, I.H. Sørensen, The B-method, in: Formal Software Development Methods, in: LNCS, vol. 552, Springer, 1991, pp. 398–405.
[10] C. Hansen, E. Syriani, L. Lúcio, Towards controlling refinements of statecharts, in: Poster Proceedings of 6th Conference on Software Language Engineering, 2013, arXiv:1503.07266 of CoRR.
[11] D. Harel, A. Naamad, The STATEMATE semantics of statecharts, ACM Trans. Softw. Eng. Methodol. 5 (1996) 293–333.
[12] B. Meyer, Reusable Software: The Base Object-Oriented Component Libraries, Prentice-Hall, 1994.
[13] B. Liskov, Data abstraction and hierarchy, SIGPLAN Not. 23 (1987) 17–34.
[14] G. Castagna, Covariance and contravariance: conflict without a cause, ACM Trans. Program. Lang. Syst. 17 (1995) 431–447.
[15] P.C. Masiero, J.C. Maldonado, I.G. Boaventura, A reachability tree for statecharts and analysis of some properties, Inf. Softw. Technol. 36 (1994) 615–624.
[16] Q. Long, Z. Qiu, S. Qin, The equivalence of statecharts, in: Formal Methods and Software Engineering, in: LNCS, vol. 2885, Springer, 2003, pp. 125–143.
[17] E. Syriani, H. Vangheluwe, R. Mannadiar, C. Hansen, S. Van Mierlo, H. Ergin, AToMPM: a web-based modeling environment, in: MODELS'13: Invited Talks, Demonstration Session, Poster Session and ACM Student Research Competition, vol. 1115, 2013, pp. 21–25, http://CEUR-WS.org.
[18] M. Crane, J. Dingel, UML vs. classical vs. rhapsody statecharts: not all models are created equal, Softw. Syst. Model. 6 (2007) 415–435.
[19] M. Bagherzadeh, N. Hili, J. Dingel, Model-level, platform-independent debugging in the context of the model-driven development of real-time systems, in: Foundations of Software Engineering, ACM, 2017, pp. 419–430.
[20] R. Milner, Communication and Concurrency, PHI Series in Computer Science, Prentice Hall, 1989.
[21] L. Aceto, M. Hennessy, Adding action refinement to a finite process algebra, in: Automata, Languages and Programming, 18th International Colloquium, ICALP91, Madrid, Spain, July 8–12, 1991, Proceedings, 1991, pp. 506–519.
[22] L.S. Barbosa, S. Meng, UML model refactoring as refinement: a coalgebraic perspective, in: Symbolic and Numeric Algorithms for Scientific Computing, 2008, pp. 340–347.
[23] E. Mikk, Y. Lakhnechi, M. Siegel, Hierarchical automata as model for statecharts, in: Advances in Computing Science, in: LNCS, vol. 1345, Springer, 1997, pp. 181–196.
[24] C. Klein, C. Prehofer, B. Rumpe, Feature specification and refinement with state transition diagrams, in: Workshop on Feature Interactions in Telecommunications Networks and Distributed System, IOS Press, 1997, pp. 284–297.
[25] P. Scholz, A refinement calculus for statecharts, in: Fundamental Approaches to Software Engineering, in: LNCS, vol. 1382, Springer, 1998, pp. 285–301.
[26] B. Paech, B. Rumpe, A new concept of refinement used for behaviour modelling with automata, in: FME '94: Industrial Benefit of Formal Methods, Second International Symposium of Formal Methods Europe, Barcelona, Spain, October 24–18, 1994, Proceedings, 1994, pp. 154–174.
[27] MathWorks, Simulink User's Guide, r2013b ed., 2013.
[28] MathWorks, Stateflow User's Guide, r2013b ed., 2013.
[29] A. Miyazawa, A. Cavalcanti, Refinement-oriented models of Stateflow charts, Sci. Comput. Program. 77 (2012) 1151–1177.
[30] IBM, Rational Rhapsody, 8.3.0 ed., https://goo.gl/7fqS15, 2018 (Accessed 30 July 2018).
[31] O.M. Group, Unified Modeling Language Superstructure, 2.5.1 ed., 2017.
[32] L. de Alfaro, T.A. Henzinger, Interface automata, SIGSOFT Softw. Eng. Notes 26 (2001) 109–120.
[33] M. Famelis, R. Salay, M. Chechik, Partial models: towards modeling and reasoning with uncertainty, in: International Conference on Software Engineering, IEEE Press, 2012, pp. 573–583.
[34] J. Padberg, M. Urbásek, Rule-based refinement of Petri nets: a survey, in: Petri Net Technology for Communication-Based Systems, in: LNCS, vol. 2472, Springer, 2003, pp. 161–196.
[35] J. Padberg, M. Gajewsky, C. Ermel, Rule-based refinement of high-level nets preserving safety properties, Sci. Comput. Program. 40 (2001) 97–118.
[36] G.A. Lewis, Incremental Specification and Analysis in the Context of Coloured Petri Nets, Ph.D. thesis, University of Tasmania, 2002.
[37] H. Huang, L. Jiao, Property-Preserving Petri Net Process Algebra in Software Engineering, World Scientific Pub., 2012.
[38] L. Lúcio, E. Syriani, M. Amrani, Q. Zhang, Invariant preservation in iterative modeling, in: Model Evolution, ACM, 2012.
[39] ITU-T, Specification and Description Language, Z.100 ed., 2016.
[40] B. Selic, A short catalogue of abstraction patterns for model-based software engineering, Int. J. Softw. Inform. 5 (2011) 313–334.
[41] M. Broy, M. Fuchs, The Design of Distributed Systems – An Introduction to FOCUS, Technical Report, Institut für Informatik – Technische Universität München, 1992.
[42] V. Aravantinos, S. Voss, S. Teufl, F. Hölzl, B. Schätz, Autofocus 3: tooling concepts for seamless, model-based development of embedded systems, in: Workshop on Model-Based Architecting of Cyber-Physical and Embedded Systems, 2015, pp. 19–26.