Controlling Refinements of Statecharts

Conner Hansen[1], Eugene Syriani[1], and Levi Lucio[2]

[1] University of Alabama, Tuscaloosa AL, U.S.A.
[2] McGill University, Montreal, Canada
chansen@crimson.ua.edu,esyriani@cs.ua.edu,levi@cs.mcgill.ca

Abstract. In incremental development strategies, modelers frequently refine Statecharts models to satisfy requirements and changes. Although several solutions exist to the problem of Statecharts refinement, they provide such levels of freedom that a statechart cannot make assumptions or guarantees about its future structure. In this paper, we propose a set of bounding rules to limit the allowable Statecharts refinement operations such that certain assumptions will hold.

1 Introduction

The Statecharts formalism is considered a defacto standard for modeling reactive systems [1]. As in any modern software development, statecharts are developed incrementally. Each incremental modification, within bounds of the rules defined in this paper, can be viewed as a *refinement* of the previous version. Therefore from a reusability and maintenance perspective, it is crucial to control how modelers can refine statechart models. UML [2] and Rhapsody [3] have addressed the topic of Statecharts refinement, but in a totally different context of class inheritance which we consider as specialization.

The goal of our work is to provide a built-in mechanism to assist modelers with incrementally designing statecharts. The refinement approach we propose abides to basic software engineering design principles, such as the *open-closed principle* stating that the model should be "open for extension, but closed for modification". We therefore explore the concept of Statecharts refinement as an analog for extension within the Object-Oriented (OO) paradigm. Within the OO world, the class-subclass structure that results from the associated extension rules comes with a certain set of expectations. For instance, if there are unimplemented functions in the superclass, then a subclass must implement them if it is to be usable. In situations where the instance of the superclass is expected, any subclass (or descendent) instance must be able to stand in for that expectation as stipulated by the Liskov *substitutability principle* [4].

Our main motivation is to provide a set of rules that govern Statecharts refinement with the intention of preserving the external behavior of the original statechart. Focusing on this preservation has the potential to increase the predictability of the to-be-refined statecharts, as well as respecting the expectations of the original statechart. Therefore we consider the refinement process to be purely additive: *i.e.,* no removal of elements from the original is allowed. Refinement is not meant as a replacement for editing which follows an entirely distinct process that should be performed on the original statechart rather than on the refined. These rules must also be focused on providing realistic and usable boundaries, though empirical studies will have to be done at a later time.

2 Refinement Relation

As defined in [1], a statechart model is defined by $SC = \langle S, T, E, V, P, g, en, ex, in \rangle$. It consists of a set of states S, transitions T, events E, and variables V. P is the set of predicates built on type values, type variables, and boolean negation and conjunction. There are two kinds of events: external events are provided as stimuli from the context outside the statechart and internal events are signals generated by the statechart. States can be basic, pseudo, OR or AND. $in : S \to \wp(S)$ denotes the containment relationship which must be non-empty for OR-states and AND-states. Non-pseudo states can define entry and exit actions $en, ex : S \to V$ over the set of variables. Transitions $T : S \times E \times \wp(E) \to \wp(S)$ define the evolution between states. They can be guarded $g : T \to P$ by a predicate over variables, triggered by events, and output event signals. As defined in [5], a configuration is a legal snapshot of the statechart before or after a microstep.

Statechart refinement is a restricted form of modification of the original model such that design decisions are preserved in the refined model. This is ensured by the refinement conditions in definition 1. To define the refinement relation between two statechart models, we consider their flattened versions using the Statemate semantics [6,7]. We denote the flattened version of SC by $flat(SC) = \langle flat(S), flat(T), E, V, P, g, en, ex \rangle$, where $flat(S)$ consists of basic states only and $flat(T) : flat(S) \times E \times \wp(E) \to flat(S)$. In the following, we denote SC_o and SC_r respectively the original and refined flattened statecharts.

Definition 1 (Refinement Relation) *The refinement $R : SC_o \to SC_r$ includes the relation $R \subseteq S_o \times S_r \cup S_o \times (S_r \cup T_r) \cup T_o \times T_r \cup T_o \times (S_r \cup T_r)$, such that the following refinement conditions are satisfied. We denote the set of all refinements for all statecharts as $\Re$.*

Inverse Surjection $\forall st_r \in S_r \cup T_r, \exists! st_o \in S_o \cup T_o : R^{-1}(st_r) = st_o$

Event and Variable Inclusion $E_o \subseteq E_r \wedge V_o \subseteq V_r$

Guard Inclusion $\forall t_r \in T_r, \exists t_o \in T_o : g_r(t_r) = g_o(t_o) \wedge p$, *for* $p \in P_r$

Structural Inclusion *:*

Let $LTS(\langle S, T, E, V, P, g, en, ex \rangle) = \langle flat(S), flat(T), E, P \rangle$.

If $s_o \xrightarrow{e[p_o]/x} s'_o \in LTS(SC_o)$, *then* $\forall (s_o, s_r) \in R \wedge s_r \in S_r :$

$s_r \xrightarrow{e[p_r]} s''_r \xrightarrow{*} s'''_r \xrightarrow{[p_r]/x} s'_r \in LTS(SC_r)^*$ *or* $s_r \xrightarrow{e[p_r]/x} s'_r \in LTS(SC_r)$

where $s_o, s'_o \in S_o, s_r, s'_r, s''_r, s'''_r \in S_r, p_o, p_r \in P$

The first condition ensures that any state or transition in SC_r is mapped to exactly one state or transition in SC_o. Therefore the inverse refinement R^{-1} is a surjection. The second condition ensures that all original events and variables must be preserved at minimum. The third condition guarantees that at least the original guard is preserved. A consequence of the fourth condition is that for any states $s_o, s'_o \in S_o$, if there is a path from s_o to s'_o in SC_o, then there must be a path from $R(s_o)$ to $R(s'_o)$ in SC_r. Note that $LTS(SC)^*$ is the transitive closure of the paths in $LTS(SC)$. We do not consider intermediate guards for the structural inclusion to allow design choices in the implementation, such as conjunction, modification, or preservation of guards. Although this is abstracted in our formalization, this may lead to certain runs of execution not being achievable.

When the above conditions are met, we say that SC_r preserves the *external behavior* of SC_o. An example of this preservation, is that if the current configuration of SC_r is in a state A and it receives an event b, if the original behavior would end the microstep in state B (and ignores all other events), then the refined behavior must also end in state B even if the sequence of events received has new, intermediate events. A substate of B is acceptable as well if B has also been refined. Consequently, the Statechart refinement allows for the following possible modifications: new events can be interjected between events from the original (*i.e.,* new microsteps 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 incremental modeling.

A notable property of R is that it is reflexive: a statechart can be refined into itself (or a copy of itself), in which case $SC_r = SC_o$. This allows one to refine only parts of a statechart while keeping others intact as in the original. R is also transitive: if SC_2 is a refinement of SC_1 and SC_3 is a refinement of a SC_2, then SC_3 is a refinement of SC_1. This enables modelers to define multi-step refinements and therefore allows for incremental modeling of Statecharts.

3 Statecharts Refinement

In the previous section, we laid out the requirements for refinement rules. In this section, we extend Statecharts with refinement annotations and define the rules that allow the developer to produce a well-formed refined statechart. These rules are intended to be statically validated, though some requirements are either difficult to or cannot be validated statically.

3.1 Extending Statecharts with Refinement Modifiers

In Statemate, Rhapsody, or UML State Machines formalisms, there is currently no way to identify a Statecharts element as needing further information added before it is ready to be utilized. This requires those who design and know the system to track what components need further details and which are ready for operation. We propose four modifiers, three of which can be applied as stereotypes to states, pseudo-states, and transitions. The fourth modifier can only be applied automatically by the tool as a stereotype to AND- or OR-states and must be combined with another modifier. Note that the modifiers are annotations to the statechart and, without altering its behavior, simply indicate elements that are possible or required to be refined to the modeler. The statechart is still executable, since it must still be well-formed regardless of the presence of modifiers.

Abstract An «`abstract`» modifier identifies a state (basic, OR or AND) or transition that requires implementation details in order to be usable. As with an abstract class within an OO language, this modifier is intended to convey clearly to the user of the statechart that the particular element in question must be refined before it can be accessed and run. As with abstract classes in OO languages, abstract states and transitions can themselves be fully implemented with no restriction. Any abstract element within a statechart must be refined into a non-abstract element when refining the statechart in order for it to be considered fully implemented.

Standard A «standard» modifier identifies a Statecharts element that can be considered implemented. This modifier allows for refinement, though any details present within the original elements must be preserved.

Locked A «locked» modifier identifies a Statecharts element that is considered implemented and non-refinable. This modifier is analogous to the *final* [8] or *sealed* [9] modifiers in modern OO languages. Therefore, when refining a statechart with locked elements, those elements will be copied as is in the refined model.

Virtual A «virtual» modifier is a pseudo-modifier that is reserved by the tool. This indicates a state that was both automatically generated by the tool and may require additional details outside the bounds of what standard refinement allows for. This modifier cannot be applied to transitions, as transitions cannot act as containers for other elements. This modifier is added to a basic state that refines into an AND- or OR-state, to any newly created orthogonal components inside the refinement of an AND-state or inside the refinement of an OR-state to an AND-state. Entirely new transitions can be added between states that already exist within the marked state, so long as the source and target of the transition are within the «virtual» state. New states can also be added, so long as the resulting addition does not create an invalid statechart. This is provided in order to cover those situations where significant structural freedoms must be granted to (potentially many) future refinements in order to maximize re-usability. This does not break the external behavior of the statechart, as these freedoms are only granted within the «virtual» state.

The «virtual» modifier is applied orthogonally alongside «abstract» and «standard»: this gives rise to «abstract, virtual» and «standard, virtual» modifiers, respectively. Their meaning is exactly the same as the respective modifier without the «virtual» modifier, only now with the additional freedoms that «virtual» grants. Note that the «locked, virtual» combination is meaningless. Since all refinements must preserve the «locked» modifier, unless explicitly stated otherwise, any substate that overrides the originals «locked» modifier will also override the «virtual» modifier. This allows for «abstract» states and transitions to exist within an otherwise «locked» state, while also preventing those states from granting themselves the «virtual» freedoms.

Modifier Application and Propagation All Statecharts elements have a modifier applied to them, though this application need not be explicit. We assume that the default Statecharts modifier is «standard» if no modifiers are explicitly applied. The modifier of a composite element (OR- or AND-state) is implicitly propagated to its contained elements, unless an inner element is assigned another modifier explicitly. This is intended to reduce the number of modifiers in the diagrams. Additionally, we view this as a more sensible development approach by requiring exceptions to be explicitly marked. This way, a specific subelement of the hierarchy of a «locked» composite element can be explicitly made «abstract» or «standard», while leaving the locked behavior of the rest of the composite element and its inner elements intact. These design decisions were made so that any tool implementing the refinement method we define will be able to utilize statechart models that were not defined with these modifiers in mind.

Table 1 illustrates the permissible refinements of modifiers. The «virtual» modifier can only be applied to refinements of an already «virtual» state, newly created

		Original				
		Abstract	**Abstract-Virtual**	**Std**	**Std-Virtual**	**Locked**
Refines to	**Abstract**	Yes	Yes	No	No	No
	Abstract-Virtual	By tool	Yes	No	No	No
	Std	Yes	Yes	Yes	Yes	No
	Std-Virtual	By tool	Yes	By tool	Yes	No
	Locked	Yes	Yes	Yes	Yes	Yes

Table 1. Valid refinements of modifiers. Std stands for "Standard".

states, orthogonal components, or the refinement of a basic state to a OR-state. Otherwise, no state can be refined to a less certain modifier as solutions should only become more concrete through refinement. Any substates of an AND- or OR-state that do not have an explicitly marked modifier will inherit the modifier of its super state.

3.2 Refinement Rules

In what follows, we describe the rules governing the refinement of Statecharts.

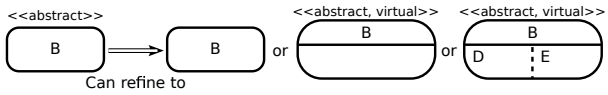

Fig. 1. The generic basic state refinement.

R1: Basic State In Fig. 1, we show an overview of the three possibilities that a basic state can be refined into: a basic state, an OR-state, or an AND-state. In the case of refinement into an OR- or AND-state, all newly created regions are given the «`virtual`» modifier by default. The modifier of the resulting state must follow the rules in Table 1.

Any defined actions of the source model are copied over to the refined state by default. Each of these actions can be modified as long as it satisfies the principle of closed behavior from OO software. We cannot verify the weakening of the pre-conditions or the strengthening of the post-conditions of an action, however it should be considered well-formed if these conditions are met. Additionally, all types of states (pseudo-states included) must have their names and in/output transitions preserved across a refinement.

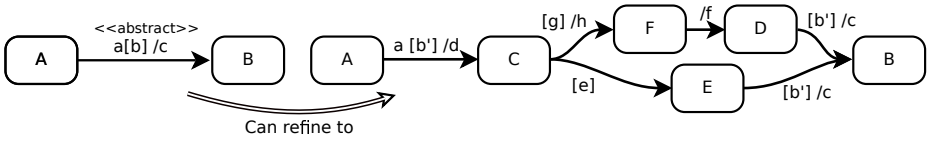

Fig. 2. The generic transition state refinement.

R2: Transitions In Fig. 2 we demonstrate a possible transition refinement. Any number of states can be inserted in between the transition's source and target states, however the execution flow through these states must pass from the source to the target as it would have in the original statechart. The only exception to this case is when a composite

element containing the source state of this transition (if such a container exists) has an outgoing transition of its own triggered, thereby prematurely ending the execution flow. This exception is practical, as forbidding this behavior would be very difficult to enforce without significantly hindering the implementor when using Statemate's *outer-first* semantics [10].

R2.1: Events The original event must be preserved on the first outgoing transition. In Fig. 2, the original event, a, is preserved after the refinement operation. Any intermediate transitions must be executed as microsteps, leading to an arrival at the original target state at the end of the processing of that particular trigger (plus the additional intermediate microsteps inserted by the refinement).

R2.2: Guard The original guard of the transition must be preserved at a minimum, though new conjunctive statements are allowed. This preserves, at a maximum, the original guard's state space. For example, in Fig. 2, guard b is refined into guard b', where either $b' = b$ or $b' = b \wedge p$ for some predicate $p \in P$. Disjunctions are not allowed since that can expand the state space without limit effectively allowing for the removal of the guard.

R2.3: Broadcast Broadcasts can be added to transitions created during refinement, excepting transitions connected to the original transition's target state (such as the new transitions from D to B and from E to B in Fig. 2). For those transitions, the original broadcast—or lack of a broadcast—must be preserved so that the behavior expected by the original target state remains preserved. In Fig. 2, the original broadcast c is preserved along the final transitions of the refinement. This is to preserve any expectation of the original statechart that a broadcast will be sent out just before the arrival at the destination state. Note that, as stated in R2, events `h` or `f` may interrupt the execution flow to state `B`.

R2.4: Mutually Exclusive Internal Guards Any guards added to transitions that are not connected to the original source or target states must have a mutually exclusive alternative path so that the execution flow is guaranteed. That is, the original reachability is preserved in compliance with Definition 1. In Fig. 2, there are two potential paths that could be taken from the added state C. The transitions have guards g and e, respectively. Therefore, the relation $g = \neg e$ must hold so that there will always be a transition that can be activated. For our implementation, we can only verify the literal negation of the guard. Future implementations would need to be more logically flexible.

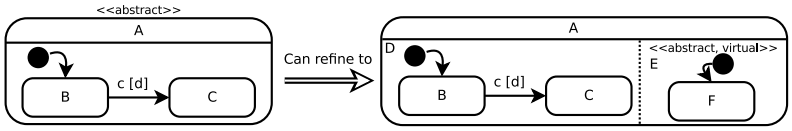

Fig. 3. Refinement of an OR-state into an AND-state.

R3: OR-State An OR-state may be refined into an AND-state with two orthogonal components where one orthogonal component contains all of the original inner elements and the other only holds a region marked as «`abstract, virtual`». Otherwise, the refinement is propagated to the inner elements of the OR-state as in Fig. 3. The dashed boundary around the internal states is meant to convey that all of the elements encompassed each have the same modifier applied.

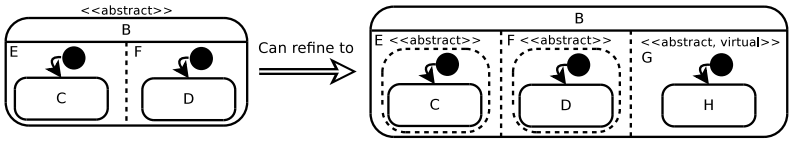

Fig. 4. The generic AND-state refinement.

R4: AND-state In Fig. 4 we demonstrate the refinement of an AND-state. As with OR-states, the refinement of an orthogonal component can be propagated to its inner elements. An AND-state can also be refined to include additional orthogonal components, which are granted the «`abstract, virtual`» modifier by default.
R5: Default State Default states follow the same rules that apply to the type of state that it is. Additional the resulting state must still be marked as default.
R6: Final State and Fork, Split, Merge, Join Pseudo-States These states are not directly refinable. The incoming and outgoing transitions must be preserved, though if the state is not «`locked`» additional incoming or outgoing transitions may be added in accordance with the modifiers applied. Final states do not allow for outgoing transitions.
R7: History State A shallow history state is either preserved or is refined into a deep history state. Deep history states are considered locked by default in order to preserve potential dependencies in the statechart.

3.3 Incremental Statechart Modeling Example

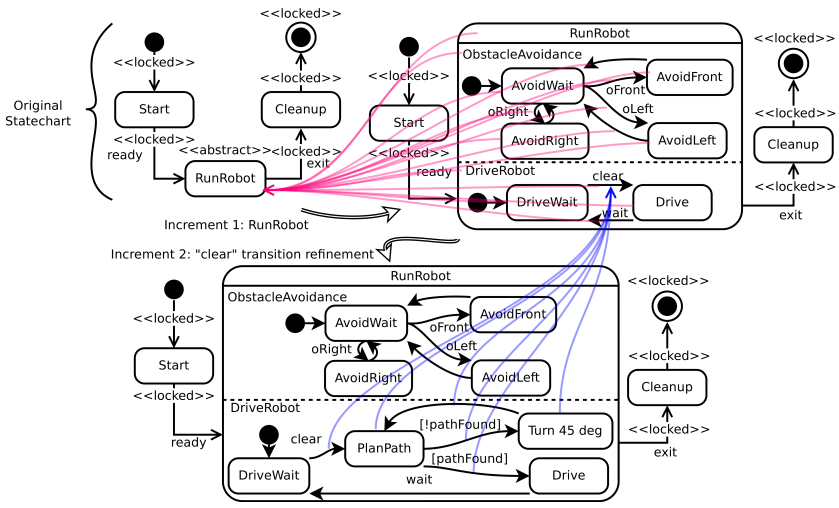

Fig. 5. The SimpleRobot statechart refinement example

To illustrate the usability of the rules governing the refinement of Statecharts, consider the initial statechart in Fig. 5 that models a generic design for robotics interface. This structure guarantees the basic start/stop behavior of anything relying on this statechart for its functionality. Since our rules rely on Statemate semantics, the outer transitions will always fire if able before any substate transitions. The only refinable element

is the RunRobot abstract basic state which is to be refined in subsequent increments. Fig. 5 also shows two increments of the initial statechart. The pink and blue arrows represent mappings of refined elements to their original elements for each refinement.

The first refinement step applies to the RunRobot basic state, which refines to an implicit «standard» AND-state with two orthogonal components. At the beginning of this refinement step, RunRobot would have had a «virtual» modifier attached to it, though this modifier was removed at the end of the refinement. This was done as an example to demonstrate that because we want certain structural guarantees, we are granting only basic freedoms to future refinements. At this stage, the DriveRobot orthogonal component models the movement of the robot while avoiding obstacles. The algorithm in this increment relies on the sensor array to return a clear event, at which point the robot will simply begin driving forward until obstacle avoidance is triggered.

A second increment models explictly the obstacle avoidance algorithm in the statechart. In this case, the transition triggered by the event clear is refined into multiple transitions and states. These states are used to do very basic, naïve path planning starting from a waiting state. This implementation allows for the robot to now plan a path before attempting to drive forward. This allows us to implement a navigational algorithm inside the DriveRobot orthogonal component, while still respecting the external entry and exit behavior. That is, if the state DriveWait receives the event clear, then the token of execution will move forward until stopping at state Drive. If state Drive receives event wait, then the execution token will, as expected, arrive back at state DriveWait. Likewise, if the exit event was to be broadcast to this Statechart, no matter how we implement the refinement detail, the RunRobot state will exit as expected.

We have implemented this example as a proof of concept of our rules for refinement in AToMPM. The initial statechart is first modeled and annotated with modifiers. Although this original statechart is executable, it was designed with the intention of being further refined at a later stage, specifically in the RunRobot basic state. This example also illustrates the need for multi-step refinements. With the current implementation, if a transition is desired to be refined into two transitions with an AND- or OR-state in between (such as if we were refining Fig. 2 and wanted state C to be an AND-state instead of basic) then this must be performed in two refinements.

4 Related Work

To the best of our knowledge, the literature has not explored the concept of Statecharts refinement in the context of incremental modeling. The closest work has been done in μ−Charts [11,12] using refinement calculus which we discuss later. Instead, it is done in the context of class inheritance, where a subclass can inherit from a superclass' statechart. We focus on three main Statecharts implementations: Rhapsody, UML State Machines, and Statemate. Crane *et al.*[13] have previously compared the semantics of these three variants, but did not discuss how they implement refinement.

Rhapsody [3] defines three types of modifiers on that can be annotated on a statechart: *inherited*, *overridden*, and *regular*.

Modifiers and Inheritance Elements marked as being inherited will allow for changes to the parent (original) statechart to propagate to the child (refined) statechart Elements

marked as being overridden no longer accept general changes from the parent statechart other than deletions. We do not provide this behavior. The original-refined relationship can be severed, leaving the refined statechart in an independent state from the original. Rhapsody additionally allows for both transitions and states to be added freely, which we only allow in states marked as «`virtual`».

Transitions In Rhapsody, events must be preserved, though the guards and any broadcast events attached to the transition can be modified arbitrarily. We do not allow this as this would allow for the behavior of the statechart to be altered arbitrarily. If a guard is desired to be removed, then either a different original statechart model should be used or it may need to be redesigned.

States Rhapsody allows one to freely override and redefine actions on refined states. Since modifying actions does not alter the external behavior of the original statechart, we also allow for free modification of actions.

The **Unified Modeling Language (UML)** presents three sets of inheritance rules for State Machines [2]: *subtyping*, *strict inheritance*, and *general refinement*. We are only concerned here with the first two sets of rules. The focus of subtyping is to preserve the pre-/post-conditions and general behavior of a state machine, such that the refined state machine can stand in place of its parent. This is aligned with our intentions.

Transition subtyping UML allows for refined transitions to change their target state, which we disallow due to this breaking the external behavior of the original statechart. As in our rules, events and sequences of events must be preserved. Guards are allowed to utilize disjunctions, which weakens the pre-condition required for the transition to be able to fire. We only allow conjunctions, which strengthens the pre-condition and remains within the state space of the original statechart. New transitions can be added as desired, which we only allow in «`virtual`» states.

States subtyping A state can add more outgoing transitions, though it does not have to preserve its incoming transitions. In our solution, all transitions must be at least partially preserved. This guarantees the certain behaviors of the original statechart. In UML, all OR- and AND-states are granted the freedoms that we only grant in «`virtual`» states.

The focus of strict inheritance is to preserve re-use associated with inheritance.

Transition strict inheritance The target, source, events, and sequences of events of a transition may be changed. We do not allow for transitions to change their source or target states, and all events must be retained in order to preserve the external behavior of the original statechart. As in Rhapsody, guards can be freely modified, which is not allowed in our solution.

State strict inheritance States preserve their outgoing transitions, while allowing for more to be added. In our implementation, only states inside a «`virtual`» state can have more transitions added. As in our solution, new states can be added to OR- and AND-states within «`virtual`» states.

Strict inheritance, as with subtyping, provides much more freedom than our solution. Our focus is to provide more predictability and clearly defined expectations of any refined statechart than the rules provided by UML.

Harel's **Statemate** [10] also addresses the concept of Statecharts inheritance, laying out several basic rules for states and transitions. In Harel's solution, basic states can be refined into other basic states, OR-states, or AND-states. Orthogonal components

can also be added to any state, though all OR- and AND-states are granted the same freedoms as our «virtual» states.

Transitions can be added as desired, whereas we require refined transitions to maintain the overall pattern of connectivity of the original transition. Statemate also allows for the target state of a transition and the guards on that transition to be changed as desired. New broadcasts can be added, though original broadcasts must be preserved in order, allowing for developers to easily break the external behavior of the original statechart.

The **μ-Charts formalism** [11,12] was introduced as a variant of Statecharts that sought to provide additional expressive power. In these papers, Scholz defines a calculus for refining μ-Charts as part of an incremental design process.

Transitions Transitions can be added so long as the event associated with the new transition does not conflict with any other transition. Our solution allows for transitions to be added only in «virtual» states. This allows limited expectations to hold. Transitions can be removed only if there is another transition present that has the same event. Guards can be added, so long as several formal conditions hold. We do not allow for events to be modified in our solution.

States The μ-Charts rules allow for the removal of initial states, which effectively amounts to the removal of an orthogonal component. We do not allow for the removal of orthogonal components. Additional states can also be added, so long as they are plugged into the existing flow using new transitions. We only allow this behavior in «virtual» states.

5 Conclusion and Future Work

In this paper, we presented a new refinement relation to support the control of further Statecharts refinements. Unlike other approaches, we restrict refinements to guarantee the preservation of the external behavior 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, such as the introduction of refinement modifiers in the Statecharts formalism. We formally demonstrated the soundness of these rules and proved the reflexivity and transitivity of the refinement relation, thus allowing for multi-step refinement. We implemented and tested our refinement rules on an example of incremental Statecharts development, in order to demonstrate how the original external behavior is preserved while also granting extensive developmental freedoms. 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 during the development life cycle.

For future work, we plan to review our design decisions to take into consideration alternative Statecharts semantics, since we currently only entertain the usage of Statemate semantics. Additionally, this method of refinement should be compared with a method driven by model transformations, which may provide additional advantages. Finally, decisions that were made with practicality in mind should be studied more extensively through implementation and user studies in order to determine edge cases associated with our approach.

References

1. Harel, D.: Statecharts: A visual formalism for complex systems. Science of Computer Programming **8**(3) (jun 1987) 231–274
2. OMG: Omg unified modeling language specification. Technical report (2001)
3. IBM: Rational Rhapsody Manual
4. Liskov, B.: Data Abstraction and Hierarchy. SIGPLAN Notices **23**(5) (jan 1987) 17–34
5. Harel, D., Naamad, A.: The statemate semantics of statecharts. ACM Trans. Softw. Eng. Methodol. **5**(4) (October 1996) 293–333
6. Harel, D., Pnueli, A., Schmidt, J., Sherman, R.: On the formal semantics of statecharts. In: Proc. 2nd IEEE Symp. on Logic in Computer Science. (1987) 54–64
7. Wasowski, A.: Flattening statecharts without explosions. SIGPLAN Not. **39**(7) (June 2004) 257–266
8. Gosling, J., Joy, B., Steele, G., Bracha, G., Buckley, A.: The Java Language Specification. 3rd edn. Addison Wesley (2005)
9. Microsoft Corporation: Microsoft C# Language Specifications. Microsoft Press (2001)
10. Harel, D., Gery, E.: Executable Object Modeling with Statecharts. Computer **30**(7) (jul 1997) 31–42
11. Scholz, P.: A refinement calculus for statecharts. In Astesiano, E., ed.: Fundamental Approaches to Software Engineering. Volume 1382 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (1998) 285–301
12. Scholz, P.: Incremental design of statechart specifications. Science of Computer Programming **40**(1) (2001) 119–145
13. Crane, M., Dingel, J.: Uml vs. classical vs. rhapsody statecharts: not all models are created equal. Software & Systems Modeling **6** (2007) 415–435