Available online at www.sciencedirect.com

SciVerse ScienceDirect

Electronic Notes in Theoretical Computer Science 295 (2013) 65–78

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

Behavioral Refinement and Compatibility of Statechart Extensions

Christian Prehofer[1]

LMU München and Fraunhofer ESK München

Abstract

We compare the notions of refinement and compatibility for system models and their variants described as statecharts. Compatibility, in the sense of substitutability, means that a system can be used in any place where the original one was used. We show that existing definitions of refinement and compatibility are orthogonal, if they include an interface extension. Then, we focus on extended compatibility, which means that the system returns to a compatible behavior after some time even if the newly added features are used. Our new result shows under what conditions the usage of a newly added feature preserves behavior. More specifically, we perform a novel kind of elimination of the newly added behavior on a trace level. In this way, we can achieve extended compatibility even if the newly added features use existing input and output events, which is not possible with existing abstractions and refinement concepts.

Keywords: statecharts, behavioral refinement, semantic refinement, compatibility, model-based development

1 Introduction

In this paper, we focus on refinement and compatibility of system models and their variants represented as statechart diagrams. The idea is to start with a base model and then to add small features incrementally by adding new states and transitions. Such iterative and modular development of statechart models can be used for modeling variations and optional features, as for instance discussed in [10]. The question addressed here is compatibility of such extensions, i.e. whether the old behavior is preserved when extending a system.

There exists considerable work on semantic refinement, which is the process of adding details while preserving behavior of the original model. Preserving behavior hereby means that the (specification of the) refined module implies the original one.

In our setting, we consider observable traces of input and output behavior. Refinement is modeled as trace inclusion. This means that all traces (i.e. all possible

[1] Email: Christian.Prehofer@esk.fraunhofer.de

http://dx.doi.org/10.1016/j.entcs.2013.04.006

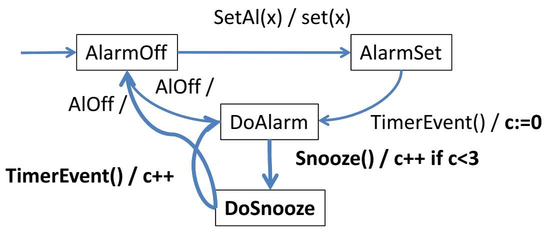

Fig. 1. Alarm extended by Snooze

behaviors) of the refined system must also be possible in the original system. Note that this is different from other work on automata, e.g. model checking, e.g. [9,5], where the internal state transitions are considered. From refinement we also like to obtain a notion of compatibility. If A is extended to A', then A' is compatible with A if we can replace A by A' in any existing implementation context and the behavior does not change. This is also called substitutability.

For instance, consider the classical stack example. Assume we have a stack data structure where we add a counter which maintains the number of current elements in the stack. Such a new feature is clearly a refinement as it preserves the behavior, and is also compatible.

Typically, refinements add internal implementation details but may also extend the input/output interface. The latter means adding new input and new output events. Abstraction or hiding is then used to relate the refined to the original system. In most cases, these abstractions simply remove these new events, see e.g. [16][11][13][4]. In case of the stack plus counter example, this yields compatibility.

In case of an interface extension, we will show that existing notions of refinement and compatibility are independent properties. Other notions of refinement only imply compatibility if new features are not used (and some other cases do not apply). Thus, we compare the notions of refinement and compatibilly in detail. Then, we develop new concepts and criteria that allow us to reason about behavior of extensions of systems even when the newly added features are used. In contrast to existing work, we do not assume that the new features only use new, distinct input and output events.

For instance, consider an alarm clock, which is extended by a snooze feature, as shown in Figure 1. By convention, we show the added elements of the new feature (here Snooze) in bold text and thicker lines. In case of an alarm, the snooze feature disables the alarm and then restarts the alarm after a short period of time. While this modifies the behavior locally, it will not affect future behavior.

Note that the new feature uses the $TimerEvent()$ input event in the DoSnooze state, which is already used in the base functionality. Hence stripping off just the new events (here Snooze) from observed traces, as done in existing refinement relations, is not enough.

In the above example, the Snooze extension changes the control flow in the statechart, but it is easy to see that the extension preserves the original behavior. Existing work essentially separates old and new behavior by distinct, new events. This is however often too limited, as in this example. In many cases, a new feature

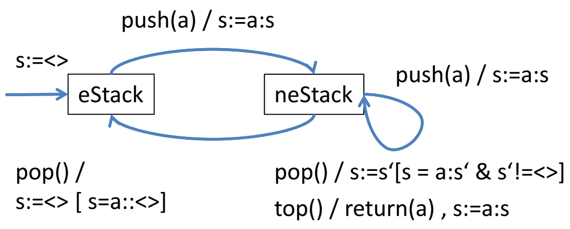

Fig. 2. Stack Example

may require input which uses existing events as in the base system, e.g. a numeric input from a standard keyboard. Similarly, assume a user should always have the option to cancel a feature by an "exit" key. Then this key should also be usable within new features, which means that simple separation into old and new events is neither desirable nor possible.

For such extensions, which use old and new events, we will show when compatibility is possible. We develop novel refinements concepts, which eliminate the effect of the newly added features by removing elements from the input and output behavior. Our main result is a precise definition of such eliminations and a new result showing when the behavior is preserved under such elimination. Even if a new feature is used several times, we can show that behavior is preserved after this usage.

Interestingly, the notions of refinement and compatibility are covered in several existing works from quite different research fields using similar, automata-based artifacts. This includes model-based development [13][8][12], model checking and formal methods [4][17], object lifecycle modeling [14], and UML modeling [15][10][16], including aspect-oriented modeling [17]. Furthermore, work on interface automata consideres a different notion of compatibility between two automata [1,5,2], focusing on input/output compatibility. The works above however do not capture the notion of compatibility and the results here.

In the following section, we introduce our statechart concepts. Then, we review and compare the concepts of compatibility and refinement for state charts where behavior is added or is even modified in Section 3. In Section 4, we introduce our new concept of refinement under elimination, where newly added features can be used while preserving behavior.

2 Statechart Model

We model software systems by statecharts, which describe the possible behavior of a software system. Note that we use a very simple form of statecharts without parallel and hierarchical states. An example is the statechart in Figure 2 describing the basic functionality of a stack. We have two states, one for the empty stack, the other for non-empty stacks.

More precisely, a statechart consists of

(i) States St, with some initial states St_i

(ii) Input events I

(iii) Output events O

(iv) Internal variables V, with initial values

(v) A transition relation $tr \subseteq \wp(St, I, V, St, O^*, V)$

We use the notation *event*() / *action*[*condition*] for transitions. A transition can be initiated by an external event, here event(). It may have a condition and it may have an action that it initiates. This action describes the behavior triggered by the transition. Note that all three labels may be empty. In case the trigger event is omitted, we have an internal transition without an external event, also called spontaneous transition. For further details, we refer to [13][8].

Our semantic model builds on the work in [13][8][12] and employs an external black-box view of the system. It is based on event traces from the outside that trigger transitions. Only the observed input and output events are considered, not the internal states. A possible run can be specified by a trace of the events and the resulting output of the statechart.

We use the loose, "chaos" semantics from [13][8][12] where the semantics of a component is given as the set of possible traces. The set of traces includes any possible trace by transitions specified in the statechart. In addition, any unspecified event, e.g. an event for which no transition is defined in the current state, leaves the statechart in chaos state and any behavior is permitted after that.

Note that our statechart model permits a non-deterministic choice if several transitions are possible in one state, which is just a special case of loose semantics. Formally, we assume traces (i,o) over finite and infinite streams over I and O, denoted as $I^\Omega = I^* \cup I^\infty$ and $O^\Omega = O^* \cup O^\infty$.

For a statechart S, we write $(s, i, o) \in S$ if there is a trace starting from state s with input i and output o. In case s is an initial state, we write just $(i, o) \in S$. Similarly, (i, o, s) denotes that the state s is reached with input i which produces output o, and (s, i, o, s') if s' can be reached from s with input i and output o.

We use the following notation on streams:

- $s :: s'$ concatenates two streams, where s is assumed to be finite.
- $a : s$ creates a stream from an element a by appending the stream s.

Furthermore, $first(s)$ is the first element of a stream s. We denote by $I \backslash In$ the elimination of elements of In from I and by $O + I$ the union of disjoint sets.

For instance, in the example of Figure 2, an input $push(5) : push(4) : top() : pop : top()$ yields the output $return(4) : return(5)$.

We assume a semantics with instant feedback [13], which means that signals sent to the statechart itself are processed instantly before new signals are taken from outside. For a comprehensive treatment of different statechart semantics see [6]. Statecharts may be non-deterministic. We call this under-specification and this leaves details open for further implementation decisions. For instance, in Figure 2, it is not specified what happens for a $top()$ event in state eStack.

Note that our semantics is total, i.e. it specifies some output for any input. Unlike other formalizations, it is not the case that behavior for some input is undefined or input is not permitted. If behavior for some input is not specified in the statechart at some state, anything is permitted (i.e. any output for any input). In this case, the statechart remains in this state. This semantics is particularly suitable for refinement and stepwise system development - for an actual implementation one has to reduce this non-determinism eventually.

3 Semantic Refinement and Compatibility

In this section, we compare the notions of refinement and compatibility. Compatibility means that a new, refined system can be used in any place where the old system was and behaves in the same way. Compatibility is here not just syntactic compatibility wrt syntactic interfaces.

For a statechart S, we say S' is an **extension** of S, if

(i) it extends the input and output events,

(ii) may add internal variables and

(iii) the behavior is modified by adding and removing both states and transitions.

Typically, an extension adds or extends some functionality, thus we speak of new features in an extension when we refer to the added functionality. In general, we permit removal of transitions, as it can be used to reduce non-determinism by eliminating an option if others exist.

We say a statechart S' is **compatible** with S if

(i) Interfaces are compatible and

(ii) S' behaves identical if used instead of S (i.e. possibly added features are not used).

In general, the term refinement is used if the original behavior is preserved under some abstraction or mapping. The following definition is typically used for refinement of traces, and will be extended later.

Assume a specification S in the form of a set of (i, o) pairs over the input events I and output O. Assume a specification S' over I' and O' which extend I and O, respectively. I.e. $I' = I + In$ and O' is $O + On$. Then S' is a **refinement** of S if $\{(i' \backslash In, o' \backslash On) \mid (i', o') \in S'\} \subseteq S$.

Basically, this definition strips the added input and output events from the extended traces of S', which then must already be possible in S. This definition is used in similarly in the existing literature, e.g. in [13][11][16] and in the form of projections in [4] and without elimination of new events in [12].

Other notions of refinement explicitly assume that each (i, o) pair with a specific behavior in S is refined and there exists a corresponding pair in S'. This is not needed in our case as the behavior is totally defined on any input. For a specific input for S, there can be several possible outputs, but at least one must be preserved or refined in S'. This is considered reduction of non-determinism.

An example for this refinement notion is shown in the extension of the stack

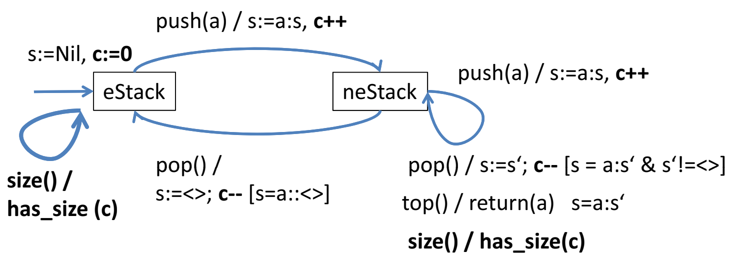

Fig. 3. Stack with Counter

by a counter, as shown in Figure 3. The counter does not modify the behavior of the stack, it only produces additional output upon the *size*() event. Hence this refinement is also compatible.

3.1 Refinement vs Compatibility

Based on the above definitions, we can now compare these two notions in more detail. The notion of refinement means adding implementation details, which should ideally imply compatibility. This is reflected in the above definition of refinement if an extension does not extend the interface, i.e. the input and output events. As our statecharts are non-deterministic but totally defined for any input, a refinement can remove options but cannot be undefined for some input. Hence, in case of identical interfaces, refinement is simple trace inclusion which implies compatibility in our setting.

Yet in many cases, see examples above, an interface extension is desirable and used in the literature. This leads to the case that both properties are independent. We will illustrate this in the following.

First, consider the example of Figure 1. It is easy to see that the extension is compatible, but it is not captured by the above definition of compatibility. If used in an existing context the *Snooze* event will not occur, and it behaves identical to the original version. The reason why the definition does not apply is the usage of old events in the extension. In case the Snooze feature is used, it may use the TimerEvent event, which is not a new event and is not eliminated in the above reduction of S' to S traces. Hence, the above definition of refinement does not apply. Other formalizations, e.g. [8][12], simply assume that usage of new feature (with new events) leads to chaos state (without the elimination of new events). Hence refinement applies, but once new features are used, no more properties hold. For this purpose, we will use the concept of extended compatibility, which permits the usage of new features.

Secondly, we should note that the above notion of refinement does not imply compatibility. An extension may add additional output of new events, even if the new input events are not used. Hence compatibility is lost as these events are not expected or undefined in the original version. A simple example is a logging feature, which does not take new inputs, but creates output events as shown in Figure 4 as an extension of the stack.

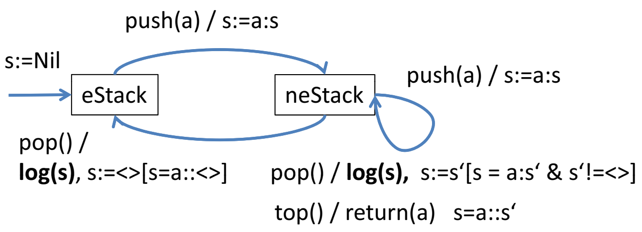
push(a) / s:=a:s
s:=Nil
eStack
neStack
push(a) / s:=a:s
pop() /
log(s), s:=<>[s=a::<>]
pop() / **log(s)**, s:=s'[s = a:s' & s'!=<>]
top() / return(a) s=a::s'

Fig. 4. Stack extended by Logging

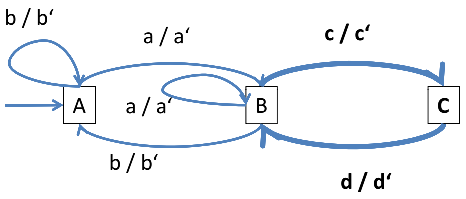

Fig. 5. Simple loopback Extension

3.2 Refinement Concepts and Non-Determinism

We discuss in the following another limitation of the above notion of refinement. The technical problem relates to our approach on chaos semantics. While chaos semantics have been shown to be very valuable to use for stepwise refinements as reduction of non-determinism [13][12], what happens if the statechart gets into chaos state while traversing the extended features.

Consider the example in Figure 5. The base statechart is in fact deterministic and fully defined over the input a and b. Consider the input $I_c = a : c : b :: x$. In this case, the extended statechart reaches state C and then goes into chaos, hence output is completely undefined and all traces are permitted. On the other hand, if we eliminate the new input event c from I$_c$, we obtain $a : b :: \ldots$, where the base automaton produces a regular, expected result. Hence the above definition of refinement, similar to definitions in [4][13], is not sufficient to express refinement here. Similar examples can be produced without chaos state –e.g. if we modify the d/d' transition in the example in Figure 5 to b/a'. Then elimination of new events on I_c yields behavior which is different from the original one.

A possible remedy is to assume that the extension is fully defined and is fully deterministic. This however does not permit successive refinement steps.

4 Eliminations for Extended Compatibility

In the following, we introduce a new technique to show compatibility using elimination of the newly added behavior on a trace level. In this way, we can achieve extended compatibility and address the limitations of existing refinement concepts.

As discussed in the last section, the definition of compatibility is not sufficient in case the newly added feature uses some of the existing input or output messages. For this purpose, we use the notion of *extended compatibility* which is defined as compatibility with the additional premise that even if a new feature is used a finite

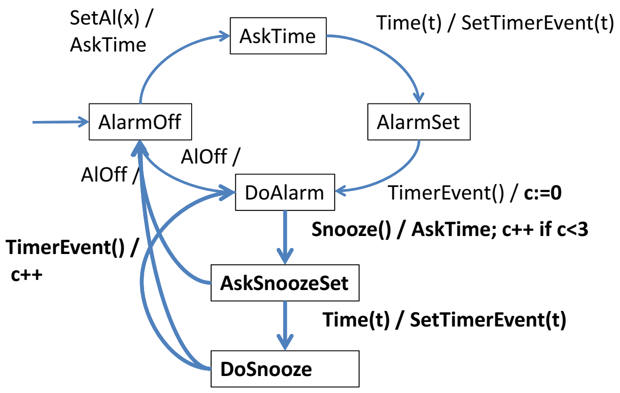

Fig. 6. Alarm extended by Flex-Snooze

number of times, the system returns to a compatible behavior after some time. More precisely in terms of traces, this means traces of the extended statechart where new input events occur finitely many times, corresponding to the usage of new features. Then eliminating the new behavior (as defined below) yields a trace of the original one. We will formalize this below.

In the following, we assume a statechart S with the events (I, O), which is extended to a statechart S' with events (In, On).

We assume here that a new feature is initiated by a new event (formalized as S'-triggered below), and then may use events from existing input and output (I, O). Typically, a new feature requires input from a keyboard, which creates an event from the existing set of events. Similarly, assume a user should always have the option to cancel a feature by an abort key. Then this key may also be used for new features, which means that simple separation into old and new events is not desirable.

An example here is Alarm plus FlexSnooze. When the alarm rings, the user can press the snooze button, followed by a keyboard entry which specifies the length of the snooze period. This is shown in Figure 6. In this example, a new feature is triggered by a new signal, but then also uses existing signals. We consider this to be a quite typical case if typical input methods and signals are reused. Examples are reading keyboard input like parameters or standard input methods like cancel or back. Here, it is not enough to abstract from the new input, here the Snooze event, as the new functionality also uses existing input.

We present in the following a formalism to eliminate the behavior of the newly added features and further show when the original behavior can be established even after the feature has been used. While it looks intuitive from the above example that this property can be established, we have to consider the following in the general case:

- It may happen that the automaton gets into chaos state while traversing the extended features. In the above example, unspecified input in the new states $AskSnoozeSet$ and $DoSnooze$ will lead to such chaotic behavior.
- The automaton may loop in the new extended feature and may not come back.

- The automaton may be non-deterministic, both in the original and extended part.

Regarding the first and second items, we observe that the extended feature is triggered by a new event, hence this can be seen as a refinement as the behavior is not specified before. However, existing definitions of refinement simply eliminate new events from the observed behavior. This is not sufficient for the cases considered here. In the following, we prepare the definition of our new notion of elimination.

We define an extension from S to S' to be **S'-triggered**, if the following properties hold:

(i) Only new states and new transitions on these states are added. (I.e. at the least start or the endpoint is a new state.)

(ii) The new transitions from old states are always triggered by new events (not in S).

S-triggered means in particular that no new transitions between old states are added, nor removal or restriction of existing transitions. This condition is important to eliminate the effect of added features - only if the newly added behavior can be recognized by a new transition, it can also be eliminated later. For instance, a logging feature which only adds logging output but no new transitions or states is not covered here. The alarm examples in Figures 1 and 6 are S'-triggered.

For an extension from S to S', we define two states s and s' to be **next-step compatible** in S, if all direct transitions from s and s' to states in S are identical. Formally, for all $i \in I$, (s, i, o'', s'') iff (s', i, o'', s'') with $s'' \in S$. For instance, in Figure 5 the states *DoAlarm*, *AskSnoozeSet* and *DoSnooze* are next-step compatible as they return via *AlOff* to the same state (*AlarmOff*) in the original statechart.

Assume a S'-triggered extension from S to S'. A predicate $el(i, o)$ over two streams i and o is called **elimination** under the following premise: $el(i, o)$ holds iff there exists a state s' in S with (s, i, o, s') such that

(i) s and s' are next-step compatible in S or $s' = s$ and

(ii) i is of the form $i = i_0 : i'$ with $i_0 \in In \backslash I$.

(iii) S' does not go into chaos state for input i on state s.

For instance in Figure 6, consider the trace T =

(SetAl() : Time(x) : TimerEvent() : Snooze() : Time(y): AlOff,
AskTime : SetTimerEvent(x) : AskTime : SetTimerEventy(y)).

The goal is to eliminate the effect of the new Flex-Snooze feature. The corresponding traversal through the old statechart is T^{el} =

(SetAl() : Time(x) : TimerEvent() : AlOff,
AskTime : SetTimerEvent(x))

In this example, we have

el(Snooze() : Time(y), AskTime : SetTimerEvent(x)),

which describes the new behavior which we aim to eliminate. Informally speaking, $el(i, o)$ holds if there is a traversal through the extension which consumes i and produces output o. The last item in the definition ensures that S' cannot go to chaos state with the input i and is needed as S' may be non-deterministic. Note that we the *AlOff* transitions from the three next-step compatible states must not be eliminated, as it occurs also in the corresponding trace of the base statechart.

If S' is an extension of S, we denote by $s|S$ the restriction of a state s to the states and variables of S. This may be undefined for states not existing in S.

We define el^* as the extension of el over a trace (i, o) where el holds and can be applied at all positions of new input events $In\backslash I$ occuring in i, from left to right in i. We write $(il, ol) \in el^*(iel, oel)$ in this case. Formally, $el^*(i :: il, o :: ol)$ holds if $i \in S$, $first(il) \in S'\backslash S$, (i, o) is a trace in S', and there exist $il1, il2, ol1, ol2$ such that $il = il1 :: il2$, $ol = ol1 :: ol2$ and both $el(il1, ol1)$ and, recursively, $el^*(il2, ol2)$ hold.

Following the above example, we have $T^{el} \in el^*(T)$ as desired. Note that el^* is partially defined, e.g. in case these new elements cannot be fully eliminated. (E.g. in the Flex-Snooze case, the trace may stop within the Flex-Snooze part, which means that the effect of this feature cannot be fully removed.)

We define **behavioral refinement under elimination** el^* from S to S' if the following holds. Assume $(i :: ils, o :: ols) \in S'$ where $i \in (In\backslash I)^*$ and there exist ie, ol such that $(il, ol) \in el^*(i, o)$ holds and $ils \in I^\Omega$. Then $(il :: ils, ol :: ols) \in S$.

This property states that S' preserves behavior on a sequence ils, after the input i which uses the new features of S'. This implies extended compatibility, as the behavior, in terms of observable traces, is fully preserved after usage of the new features.

We aim to define simple criteria in order to establish behavioral refinement. For this, we first define a single-step refinement.

We define **one-step refinement under elimination** el from S to S' if the following holds: if $(s, iel :: ils, oel :: ols) \in S'$ where $first(iel) \in In\backslash I$ and $el(iel, oel)$ holds, $ils \in I^\Omega$, and furthermore, $s|S$ is defined in S. Then $(s|S, ils, ols) \in S$.

This definition states that the elimination function removes a part of the input/output behavior without affecting the global behavior, for any finite or infinite input (on I) after the elimination. Note that el may depend on the internals of S'. Recall that our statechart semantics is totally defined. Thus, the other direction of the implication in this definition follows for one of the possible outputs of an input. Assume $(s|S, ils, ols) \in S$ and $(s, iel :: ils, oel :: ols') \in S'$ with $el(iel, oel)$. Then the above definition implies $(s|S, ils, ols') \in S$, which means that at least one possible behavior of S is preserved.

The above definition shows that a single use of such a feature does not affect later behavior. We aim to show next that the definition is strong enough to show that even after multiple usage of a feature we return to original behavior.

For this to hold, we need more assumptions on the extenstion. In particular, it is possible that an extension changes future behavior only if used more than once. Examples are simple to construct as extension can add local state.

We say S' is a **conservative** extension of S under some elimination function if an input to S' leads to the same state in S under the elimination on the input and under the restriction to S. Formally, $(s, i, o) \in S'$ if $(s|S, i', o)$ and $el(i, o) = (i', o')$.

We show that this implies property behavior for multiple usage.

Theorem 4.1 ***(extended compatibility under elimination):*** *Assume S' is a one-step refinement under elimination el from S with elimination function el and S' is a conservative, S'-triggered extension of S. Then S' is a behavioral refinement under elimination el' of S.*

Examples for this theorem are the snooze features in Figure 1 and 6. To apply the above theorem, we construct the el function from the added new feature as shown in the example above.

In this way, we can show compatibility of system variations. In more detail, we obtain a constructive check if extended compatibility holds for an extension. We can construct eliminations from the graphical representation of the added feature, then we have to check the conditions above (S'-triggered and conservative.). This can be done easily in most cases, unless the code in the conditions of a new transition contains complex programming constructs, which is usually not the case.

It is interesting to discuss why the step from a single, local modification to multiple modifications requires such strong assumptions on the added features. It is in fact easy to construct examples where extensions modify the later behavior, but not if used only once. For instance, a statechart can just record in a variable how often a feature has been used and adapt behavior to it - as for instance with the variable i in Figure 1. Hence, we require that the feature does not modify the state of the existing statecharts. This also means that we cannot show this result just by reasoning on external input and output behavior.

5 Related Work

In the following, we discuss related work on statechart refinement and related concepts like UML state machines and automata models. We claim that our concepts of refinement under elimination are new and can cover a practical class of examples where new features use existing events. Furthermore, we can also cover cases where new features add states which are undefined for some events and hence may lead to chaos behavior.

Earlier work on statechart refinement [13][8][12], which is using similar semantic models of statecharts, has developed several rules for refinement. These are expressed in terms of statechart entities like adding transitions or states. The main results are that refinement follows from simple conditions on the operation on a statechart. For instance, adding a new transition, triggered by a new event, results in a refinement. These results however do not consider compatibility nor covers our notion of extended compatibility. Specifically, the work in [13] uses the notion of refinement as above, but does not cover the form of elimination nor extended compatibility. The work in [12] is not explicit about eliminating newly added events or messages. It is assumed that in case of undefined events, the statechart goes to chaos state, which in turn does not allow to cover the compatibility after the usage of new features (as developed here).

The work [11] develops a refinement calculus for statecharts as in [13] based on a mapping to the Z language and relating it to refinement and simulation in abstract data types. The basic mechanism for these is also the elimination of new input and output events, as discussed before. Refinement with the focus on stepwise development and composition of services is covered in [4]. For automata model represented as tables, composition of services is developed. As in the above, refinement is based on abstraction of the new input and output events, which is limited as discussed earlier.

For related work on UML modeling, the concepts developed in [15] essentially cover basic cases of refining a state into several ones, which is different and not covered here. The work in [10] focuses on modeling the added features as independent and modular entities, modeled as statechart fragments. Furthermore, feature interactions are modeled as such fragments. For refinement, the concepts in [8][12] are used.

Other work on UML in [16], which builds on concepts for object lifecycle modeling [14], considers the problem of consistent inheritance and observation consistency, which are similar to our notion of compatibility. As in the work above, they consider abstraction from new input signals to show such properties. The case is however simpler, as they do not consider output events.

Other work on modularity for model checking [3][9] also considers the problem of extending automata models by new states and transitions. In these works, composition of statecharts leads to proof obligations for specific properties to maintain. These are in turn to be validated by a model checker. Hence, these approaches are quite different from the work presented here. Specifically, they require the specification and establishment of each individual property after the extension. Here, we focus on compatibility for any behavior. Similar goals have been pursed in the context of aspect-modeling for state machines, as shown in [17].

There is also recent work on compatiblity for interface automata [1,5,2]. Here, compatibility essentially means that the input and output behavior of two automata are compatible, in the sense that one automata may not send some event when the other one is not ready to receive this event. Refinement is based on alternating simulation (more input, less output) and is shown to preserve compatibility between two automata.

In other works, optional transitions in automata are marked explicitly and are called modalities [2,7]. Here, we strongly rely on a uniform notion of nondeterminism. Even though the system model is quite different as discussed, our notion of compatibility is closer to the notion of conformance in [7], based on a simulation relation between the states of two automata. This notion of conformance does however not cover the interfaces extensions by new events, as discussed here.

6 Conclusions

We have compared the notions of refinement and compatibility for system models described as statecharts. While refinement roughly states that the original behavior is preserved in an extended system (by some abstraction), compatibility means that such a system can be used in any place where the original one was used. Specifically, if new features extend the interface, these properties are independent. We have discussed under what conditions refinement implies compatibility. If an extended system produces new output events without explicit usage of new inputs, then refinement does not imply compatibility. In turn, if an added feature uses old input/output events, compatibility does not imply refinement based on simple elimination.

In many cases, added features change the behavior locally, but not the future behavior. To capture this, we have introduced the notion of extended compatibility, which extends compatibility by an additional premise: even if a newly added feature is used, the system returns to a compatible behavior after some time. Our new result shows under what conditions the usage of a new feature preserves the original behavior, even after repeated usage. Roughly speaking, we need to assume that the extension does not interfere with the base model and is triggered by new events.

Our main new result shows when extensions of a system preserve compatibility. This can be used for incremental system development as well as for managing variations of one system, as shown above. In particular, we aim to show in many cases that an extension is compatible with an existing system, which is important in many application scenarios.

The main new technical concept is the elimination of the newly added behavior on a trace level. Essentially, we extract the possible, new behavior and eliminate this from the observable traces. This is possible in our representation using statecharts, as the control flow (state transitions) is very explicit in this model. In this way, our refinement concepts can still be computed from the traversals of the added functionality, and is considerably more precise than existing relations. Whereas existing work on refinement simply eliminates newly added events in an extension (which includes an interface extension), we can handle many practical cases where the extension uses existing events.

We construct such an elimination operation from a single usage of a new feature, following the graphical representation of such a statechart extension. Based on this, we can achieve extended compatibility even if the newly added features use existing input and output events. For this result, additional assumptions are needed to infer that multiple usage of a feature preserves behavior. In particular, usage of the new feature may not modify the state of the existing feature. This leads to many new applications which are not possible with existing abstractions and refinement concepts.

For future work, we aim to cover a larger set of statechart concepts such as hierarchical and parallel composition. Secondly, an other interesting aspect is the timing behavior of such extensions, which is not considered here.

Acknowledgements. The author would like to thank Peter Scholz and Sebastian Bauer for discussion and feedback on this paper.

References

[1] L. Alfaro and T. Henzinger. Interface-based design. In M. Broy, J. Grnbauer, D. Harel, and T. Hoare, editors, *Engineering Theories of Software Intensive Systems*, volume 195 of *NATO Science Series*, pages 83–104. Springer Netherlands, 2005.

[2] S. Bauer, P. Mayer, A. Schroeder, and R. Hennicker. On weak modal compatibility, refinement, and the mio workbench. In J. Esparza and R. Majumdar, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 6015 of *Lecture Notes in Computer Science*, pages 175–189. Springer Berlin / Heidelberg, 2010.

[3] C. Blundell, K. Fisler, S. Krishnamurthi, and P. Van Hentenrvck. Parameterized interfaces for open system verification of product lines. In *Automated Software Engineering, 2004. Proceedings. 19th International Conference on*, pages 258 – 267, September 2004.

[4] M. Broy. Multifunctional software systems: Structured modeling and specification of functional requirements. *Sci. Comput. Program.*, 75:1193–1214, December 2010.

[5] A. David, K. G. Larsen, A. Legay, U. Nyman, and A. Wasowski. Timed i/o automata: a complete specification theory for real-time systems. In *Proceedings of the 13th ACM international conference on Hybrid systems: computation and control*, HSCC '10, pages 91–100, New York, NY, USA, 2010. ACM.

[6] R. Eshuis. Reconciling statechart semantics. *Science of Computer Programming*, 74(3):65 – 99, 2009.

[7] D. Fischbein, S. Uchitel, and V. Braberman. A foundation for behavioural conformance in software product line architectures. In *Proceedings of the ISSTA 2006 workshop on Role of software architecture for testing and analysis*, ROSATEA '06, pages 39–48, New York, NY, USA, 2006. ACM.

[8] C. Klein, C. Prehofer, and B. Rumpe. Feature specification and refinement with state transition diagrams. In *Fourth IEEE Workshop on Feature Interactions in Telecommunications Networks and Distributed*, pages 284–297. IOS Press, 1997.

[9] J. Liu, S. Basu, and R. Lutz. Compositional model checking of software product lines using variation point obligations. *Automated Software Engineering*, 18:39–76, 2011.

[10] C. Prehofer. Plug-and-play composition of features and feature interactions with statechart diagrams. *Software and Systems Modeling*, 3:221–234, 2004.

[11] G. Reeve and S. Reeves. Logic and refinement for charts. In *Proceedings of the 29th Australasian Computer Science Conference - Volume 48*, ACSC '06, pages 13–23, Darlinghurst, Australia, Australia, 2006. Australian Computer Society, Inc.

[12] B. Rumpe and C. Klein. Automata describing object behavior. In *Specification of Behavioral Semantics in Object-Oriented Information Modeling*, pages 265–286. Kluwer Academic Publishers, 1996.

[13] P. Scholz. Incremental design of statechart specifications. *Science of Computer Programming*, 40(1):119 – 145, 2001.

[14] M. Schrefl and M. Stumptner. Behavior-consistent specialization of object life cycles. *ACM Trans. Softw. Eng. Methodol.*, 11:92–148, January 2002.

[15] A. J. H. Simons, M. P. Stannett, K. E. Bogdanov, and W. M. L. Holcombe. W.m.l.: Plug and play safely: Rules for behavioural compatibility. In *In: Proc. 6th IASTED Int. Conf. Software Engineering and Applications*, pages 263–268, 2002.

[16] M. Stumptner and M. Schrefl. Behavior consistent inheritance in uml. pages 451–530. 2000.

[17] G. Zhang and M. Hölzl. Hila: High-level aspects for uml state machines. In S. Ghosh, editor, *Models in Software Engineering*, volume 6002 of *Lecture Notes in Computer Science*, pages 104–118. Springer Berlin / Heidelberg, 2010.