Assume-Guarantee Specifications of State Transition Diagrams for Behavioral Refinement

Christian Prehofer

LMU München and Fraunhofer ESK München,
prehofer@pst.ifi.lmu.de

Abstract. In this paper, we consider extending state transition diagrams (SDs) by new features which add new events, states and transitions. The main goal is to capture when the behavior of a state transition diagram is preserved under such an extension, which we call behavioral refinement. Our behavioral semantics is based on the observable traces of input and output events. We use assume/guarantee specifications to specify both the base SD and the extensions, where assumptions limit the permitted input streams. Based on this, we formalize behavioral refinement and also determine suitable assumptions on the input for the extended SD. We argue that existing techniques for behavioral refinement are limited by only abstracting from newly added events. Instead, we generalize this to new refinement concepts based on the elimination of the added behavior on the trace level. We introduce new refinement relations and show that properties are preserved even when the new features are added.

1 Introduction

State transition diagrams (in short: SD) are used in various forms to model software, e.g. modeling a software component which interacts with the environment based on events. In this paper, we consider behavioral models represented as state transition diagrams which are incrementally extended by new features. The main goal is to reason about the behavior and definedness of such an extended state transition diagram in a modular way.

The idea of *incremental development* is to start with a base model and then to add small features in succession, which add previously unspecified behavior. Extending an SD by a feature means to add new states and transitions.

Assuming such a (syntactic) extension of an SD, the question addressed here is whether the old behavior is preserved when incrementally extending an SD. This we call behavioral refinement. We use a behavioral semantics based on the observable traces of input and output events, respectively. Behavior preservation means that the resulting output trace is unchanged for all input streams, possibly under some abstraction.

As an example consider the lock extension in Figure 1, which adds a new *locked* state and ignores any input in the lock state except for the *unlock* event.

E.B. Johnsen and L. Petre (Eds.): IFM 2013, LNCS 7940, pp. 31–45, 2013.

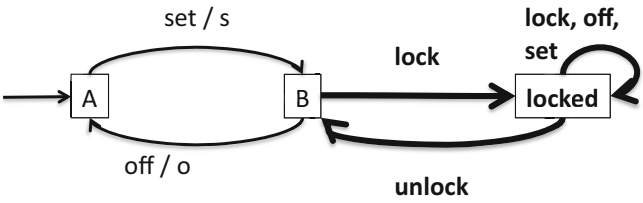

Fig. 1. Locking Feature

By convention, we show the added elements of the new feature in bold text and thicker lines.

In this example, it is easy to see that the behavior of the original base SD is preserved if no *lock* event occurs. While this is a basic compatibility property, we aim to go beyond. As we can see, even after traversing the lock extension, the SD behaves as before. However, during the traversal the externally visible behavior is altered. Furthermore, it may be the case that the SD does not return from the extension if no *unlock* event occurs. We aim to capture these observations in a formal calculus. For this, we address the following two main issues.

The first issue is that the extension also uses the *off* and *set* events of the base SD. Due to this, existing refinement and simulation techniques, e.g. [16, 17, 18, 19, 1, 14] , are not sufficient as they only abstract from newly added events, by definition. This is important in many cases when events are reused in the extension, as in this example above. For this purpose, we use a new concept which eliminates behavior on the trace level. Such an elimination essentially removes the added behavior on the level of observable input/output traces. Technically, we will use entry and exit events to detect and eliminate those segments of the trace that correspond to the newly added behavior.

The second, main point about this example is that we need assumptions on the permitted inputs, both for the base SD as well as for the extension, to reason about the behaviour. For instance, we may want that the extension always terminates and returns to the old SD. In the above example, the extended SD may loop forever in state *locked*. This can be avoided by restrictions on the permitted input. In general, both the base SD and the extension may have assumptions on the permitted input. From these two assumptions, we aim to create a single, combined assumption on the permitted input for the extended SD.

As in other assume/guarantee calculi, we use assumptions to specify what inputs are permitted. These need to make sure that the SD is defined for the permitted inputs, i.e. there is a defined transition for each event in an execution for a given input stream. Our notion of SDs is similar to interface input/output automata in [8], which use a separate state-based model to describe the input assumptions. Here, we use basic predicates to specify the input, not models. Note that our notion of automata is different from interface automata [1, 4], which are intended to specify which input events are permitted for an interface.

For instance, in the example above, we also want to ensure that the *lock* event only occurs in state B, not in state A where it is not defined. Thus, we have to reason about the permitted inputs, both in the extension and in the base SD. This is the main motivation for the assume/guarantee specifications, which are used to formalize this in a modular way.

In summary, the goal is to extend SDs by additional features with new states and transitions, and then to reason about the behavior of the extended SD. For this, we develop a notion of assume/guarantee specifications for SDs. The main idea is to make the permitted inputs and guarantees explicit on the trace level. This follows typical assume/guarantee specifications. We introduce new concepts for semantic refinement based on behavior elimination and present new results when an extension preserves behavior with respect to the base SD.

Our work is similar, on a conceptual level, to aspect- and feature-based programming languages. For these, there are results on so-called conservative extensions or observer aspects, which only add additional behavior but do not modify behavior (see [12, 5]). In other words, we aim to apply these ideas also to SDs, where we are reasoning only about input/output behavior, and not about internal state as on the programming language level. There was recent work to extend automata by aspects, as for instance [20], which includes a calculus for reasoning about automata, but does not identify specific classes of refinement and property preservation. There is earlier work on elimination based refinement in [13], which also permits non-deterministic SDs and also does not require explicit exit events. While there are first results on behavior preservation, in [13] it is not possible to reason about definedness and termination of such extensions.

The paper is organized as follows. In the next section, we introduce the syntax and semantics of SDs. Then, we define syntactic extensions on SDs, followed by behavior eliminations on the semantic level. In Section 3, we introduce new refinement concepts based on these elimination concepts. In Section 4, we present new results to show when a refinement relation can be established for an SD extension. Finally, Section 5 discusses related work, followed by conclusions.

2 State Transition Diagrams

We model software systems by SDs that describe the behavior of a software system. More precisely, an SD consists of

(i) States St, with an initial state $s_0 \in St$
(ii) Input events I and disjoint output events O
(iii) A vector of internal variables v^n, ranging over a vector of values V^n with initial values $V^{initial}$.
(iv) A transition function $tr : St \times I \times V \mapsto St \times V \times O^*$

A transition is triggered by an input event and produces a set of output events. It may have an action that it initiates. This action describes the output events triggered by the transition and the changes on the internal variables. We use the

notation *event / action* for transitions. The vector of variables describes the values of the variables and is also called variable valuation.

We focus on deterministic SDs as defined above. For the non-deterministic case, there exist several other issues, as considered in [13].

2.1 Behavioral Semantics

Our semantic model employs an external black-box view of the system. It is based on events 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 SD.

Formally, we assume traces (i, o) over finite and infinite streams over I and O, respectively, denoted as $I^\omega = I^* \cup I^\infty$ and $O^\omega = O^* \cup O^\infty$. Note that for each input event, there is a set of output events if a transition is defined. Hence for the n-th element in i, the n-th element of $S(i)$ is the corresponding set of output events.

For an SD S and a finite or infinite input stream i, we say S is **defined** for i, if there is always a defined transition for each input event in i when executing S with input i. This is written as $Def(S(i))$. For instance, in the above example, the lock SD is undefined for the input *unlock* in the initial state. We write $S(i)$ to denote the output of S for i if S is defined for i.

For a state s and a variable valuation V, we write $S(i, s, V) = (o, s', V')$ to denote the state s' and variable valuation V' after running S at state s with input i and V. This assumes that S is defined for i at state s. We also write $S(i) = (o, s, V)$ if S is run from the initial state. We write $S(i) = s$ if o and V are not of interest. Two SDs are considered equivalent if they behave equivalently for all inputs.

We denote the empty trace as *Nil* and use the following notation on traces:

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

When clear from the context, we often write just e instead of $\{e\}$ for singleton sets, and also $a : a$ instead of $a : a : Nil$. 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.

As an example, consider the alarm SD as shown in Figure 2, which is extended by a flexible snooze function called Snooze. When the alarm rings, the user can press the snooze button, which sets a new alarm (after a snooze period). In this example, observe that the extension uses new events, like *Snooze*(), but also existing ones like *AskTime*().

For instance in Figure 2, a possible trace (with input events shown above the corresponding output events) is $T =$

$(SetAl : TimerEvent : Snooze : TimerEvent : AlOff,$
$setTimer : StartAlarm : \{StopAlarm, setTimer\} : StartAlarm : StopAlarm).$

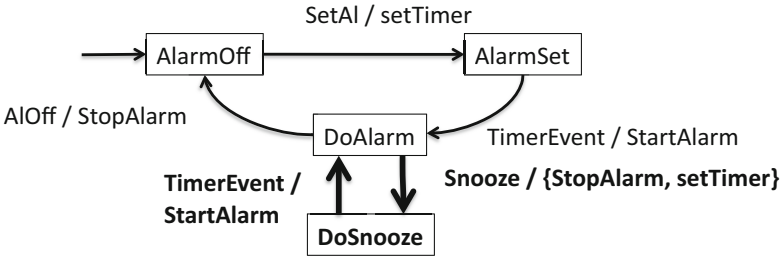

Fig. 2. Alarm extended by Snooze

2.2 Syntactic Extensions of State Transition Diagrams

When adding new features to an SDs, we use the following notion of syntactic extensions of SDs. While we permit any syntactic extensions in the definition below, this will be restricted further below to establish refinement relation on the semantical level.

For an SD S, we say S is **extended** by E to an SD S', if:

(i) S' results from adding both states and transitions on S,
(ii) S' may extend the input and output events of S, and
(iii) S' may add internal variables to S.

S' is also called an **extension** of S by E. Examples of extensions are shown via the bold states and transitions in Figures 1 and 2.

We can alternatively define extensions as a set of states and transitions to be added, which corresponds to a partial or incomplete SD. This is however more subtle, as we need to ensure that a composed SD is well defined, which is implicit here.

3 Assume-Guarantee Specifications and Refinement

In the following, we develop concepts to explicitly specify the assumptions on the input and the resulting output guarantees as in typical assume-guarantee specifications [1]. We will use predicates over finite and infinite streams. We denote assumptions as a predicate A where $A(i)$ is a Boolean value over a stream i, and predicates G as guarantees over a pair of input and output streams, $G(i, o)$.

We use the following notation for assume guarantee specifications over SDs. Assume an SD S, an assumption A and a guarantee G over streams. Then

$$A/S/G$$

states that for all input traces i where A holds, S is defined for i with $S(i) = o$, and $G(i, o)$ holds.

We also write just

$$A/S$$

which then denotes that S is defined for inputs i where $A(i)$ holds.

Assumptions can express two things, unwanted cases and unspecified cases, which we do not distinguish here. Unspecified cases are cases which shall be defined in a later phase by incremental refinement, while unwanted cases must be avoided by the environment and are not allowed.

The typical purpose of assumptions in our treatment of extensions is to specify which inputs are allowed in what phase of a traversal. For instance, when traversing an extension, we may only permit specific events.

The common notion of refinement on SDs is to allow more inputs and to produce less outputs, see for instance [1]. We can formalize more inputs easily by our notion of assumptions. We consider guarantees on the new output based on assumptions for new inputs. Note that we do not allow one to drop individual output events in the output stream.

Assuming a specification $A/S/G$, we can relax the assumptions if the guarantees hold. Also, the guarantees can be strengthened. Formally, $A'/S/G'$ is a **refinement** of $A/S/G$ if $A(i) \implies A'(i)$ and $G'(i,o) \implies G(i,o)$.

3.1 SD Extensions and Refinements

In the following, we aim to cover extensions an SD which add new features with additional behavior. The problem is now that assumptions and guarantees need to consider different input and output events over an extended interface.

For the purpose of refinement, we consider in the following equality on the output traces as follows: Assuming a specification A/S, then A'/S' is a **refinement** of A/S, if $A(i) \implies A'(i)$, and $A(i)$ implies $S(i) = S'(i)$.

This means that S and S' must behave identically for the input permitted for S', i.e. when A holds. In other words, when S' is restricted to the input for A for S, they behave the same. Internally, the two SDs may differ in states and transitions. Compared to the above assume/guarantee specifications the following holds: If A'/S' is a refinement of A/S and G and G' coincide on the inputs permitted for A, then $A'/S/G'$ is a refinement of $A/S/G$.

A typical example is an extension by a new event, after which the system may behave in a completely different way. This notion of refinement is for instance used in [1] when new events are added, but also in [7, 15], even though different formalisms are used. The main limitation here is that no guarantees hold after a new event occurs. In detail, an extension may add new events and the assumptions A do not apply for any input which contains new elements. In the lock example, an assumption predicate over the base SD only considers *set* and *off* as input events, not *lock*. Furthermore, this is only a notion of refinement and does not give any statement when the extended SD is defined.

3.2 Trace Eliminations for Added Features

In the following, we detail our approach to eliminate the behavior of the newly added features. This is used for our notion of behavioral refinement in the following sections. We first discuss important restrictions on SD extensions to determine suitable eliminations and to define the refinement relation. Then,

in the subsections below, we define the eliminations and refinement relations precisely.

We determine eliminations of added features on the behavioral level, i.e. on traces. For this, we assume that the new features are triggered by an entry event from I and return to the original SD with an exit event. In the case of the lock feature, the entry event is the *lock* event, and the exit event is the *unlock* event. Then eliminations shall remove all sequences of the form $entryEvent : \cdots : exitEvent$. We call this trace-based eliminations.

For traces over such an extended SD S' over S with extension E, we define eliminations based on entry and exit events. We say that an extension E is **entry-exit triggered**, if there are some **entry events** E_{en}, which do not occur in S. Furthermore, for each entry event $e \in E_{en}$ there is a set of **exit events** $E_{ex,e}$. This means that the states and transitions in E are only reached via some entry transition with an event $e \in E_{en}$. Furthermore, for each such entry point with e, it must be ensured that the extension returns to the original SD S if and only if an event from $E_{ex,e}$ occurs. Furthermore, we assume that it returns to the same state in S where the entry event occurs. This state is also called **join state**, similar to join points in aspect-oriented languages.

We define eliminations based on the entry and exit events as follows. Assuming E, S, and S' as above, an **elimination** el removes all trace segments of the form

$$(i_1 : \cdots : i_n, o_1 : \cdots : o_n),$$

where $i_1 \in E_{en}$, $i_n \in E_{ex,i_1}$ and $i_j \notin E_{ex,i_1}$ for $1 < j < n$. Furthermore, an infinite trace segment $(i_1 :: i, o)$ is eliminated if $i_1 \in E_{en}$ and no element from E_{ex,i_1} occurs in (i, o).

In other words, if the extension does not return, we cut off the complete, infinite part after the entry. Then, we define $el(tr)$ for a trace tr, where the elimination function el is applied from left to right over the full trace tr. This results in a finite trace if there is an entry event without a corresponding exit event.

Note that we use an elimination el in two forms. For input and output traces, we write $el(i, o) = (i', o')$. We also write $el(i) = i'$ which yields an input stream.

As an example, we continue with the above trace T for the alarm SD in Figure 2. The goal is to eliminate the effect of the new Snooze feature. The corresponding traversal through the old SD is

$$T' = (\ SetAl : TimerEvent : AlOff,$$
$$setTimer : StartAlarm : StopAlarm).$$

In this example, we have eliminated the trace segment $Snooze : TimerEvent$ and $\{StopAlarm, setTimer\} : StartAlarm$ which corresponds to the new behavior which the new feature adds. Then, we can show that the original behavior of the SD is preserved by the extended SD "modulo" the elimination.

3.3 Weak Elimination-Based Refinement

We now consider extensions which add behavior temporarily, but then return to existing, old behavior (unless they diverge). For such a case, we use eliminations

to define a refinement relation. Elimination is used to compare the input/output traces of the original and the extended SDs. It is a generalization of the typical notions of refinement which remove added behavior by removing the new events. We first define the refinement notion and then discuss its utility.

Assume S over (I, O) is extended to S' with some extension E over (I_E, O_E). The extended system A'/S' is a **weak elimination-based refinement of** A/S, if the following hold:

(i) for any stream i over I, $A(i) \implies A'(i)$ and $S(i) = S'(i)$.
(ii) for any stream i over $I \cup I_E$, if $A'(i)$ holds, then $el(i)$ is a stream over I, $A(el(i))$ and further $el(i, S'(i)) = (el(i), S(el(i)))$.

For this notion of refinement, we require that the extended SD, S', behaves as S under an elimination. We assume that for any permitted input i for S' (i.e. A' holds), the elimination on i results in a syntactically correct input for S and A holds. Otherwise, A may not be defined for $el(i)$. In other words, A' allows more input, even over an extended input event set, but additional traces must correspond to a trace of the original SD. This is enforced by the restriction that $el(i)$ is a stream over I. A possible case when an elimination may not remove all new elements not in I is when an exit event occurs before an entry event.

The notion of weak refinement essentially says that an SD behaves as before unless a new feature is traversed. It will behave as before after multiple use of a new feature, if the new features return to the original SD. Regarding properties of SDs, we can use this notion of refinement to establish safety properties as follows. Safety properties usually state that some "bad" events do not occur. If an SD S does not produce a "bad" output event b under some assumptions A and an extension E also does not produce b (possible under assumptions), then the combined system also does not produce the bad event b. A more basic, but important question is when an extended SD is defined. To establish our refinement we need to fix assumptions under which the extended SD S' is defined, considering both assumptions for the base SD S and the extension E. This will be covered in Section 4.

Compared to the analysis of different kinds of aspects considered in [5], this case is similar to observers with possible non-termination in the extension. In [5], there is also the notion of observers with abortion, i.e. termination of the program. This concept is not sufficient for our setting of SDs as traversals may remain infinitely long in an extension. Instead, we consider possible divergence and termination of the extension by assumptions, as covered in the next section.

3.4 Strong Elimination-Based Refinement

In the above notion of weak elimination-based refinement, we have assumed that the extended SD behaves as the original one under the elimination. We did not require that a traversal through the extension terminates. In case an extension of an SD is entered but the SD does not return from the extension, we only compare the finite parts of the execution. In the following, we define and discuss

a stronger notion of refinement, which requires termination for any traversal of the extension.

Assume S over (I, O) is extended to S' with some extension E over (I_E, O_E). The extended system A'/S' is a **strong elimination-based refinement of** A/S if for a stream i over $I \cup I_E$ $A'(i)$ implies the following:

(i) for any stream i over I, $A(i) \implies A'(i)$ and $S(i) = S'(i)$.
(ii) for any stream i over $I \cup I_E$, if $A'(i)$ holds, then $el(i)$ is a stream over I, $A(el(i))$ and further $el(i, S'(i)) = (el(i), S(el(i)))$. Furthermore, if i is infinite, then $el(i)$ is also infinite.

With the last clause in the definition, which is the only difference to the notion of weak refinement, we ensure that a possible extension does not diverge when entered. Clearly, this definition is only sensible if we consider infinite traces which can express divergence.

As strong elimination-based refinement entails weak elimination-based refinement, it can be used to show safety properties as above. As extensions terminate, also many liveness properties are preserved. A typical liveness property is that some (output) event o eventually occurs in all possible executions. In case this holds for the base SD, this is preserved by strong refinement. In this case, an extension may produce extra o events, but it will return to the original SD which eventually produces the o event.

4 Establishing Elimination-Based Refinements

In the following, we aim to establish refinement relations for a given base SD S with an extension E. Based on assumptions for S and E, we show that there exist specific assumptions under which an extension E is an elimination-based refinement. This also serves to reason modularly about extensions of SDs based on properties and assumptions for the SD and the extension. As discussed above, weak elimination-based refinement is suitable for safety properties, while strong elimination-based refinement can also be used for liveness properties.

So far, we have defined assumptions for the inputs of a normal SD. Next we define assumptions specifically for extensions of an SD, which only cover the input events when traversing the added transitions and states in an extension. In other words, we restrict the input while the SD is in the traversal of an extension.

For this purpose, we first generalise assumptions to specific states of an SD. We write $A(s, i)$ for A to hold at some state s with input i. Thus, the above $A(i)$ means that $A(s_o, i)$ for A to hold in the initial state.

Assume S' is an entry-exit triggered extension of S by E. Then for a predicate AE on inputs streams, we denote the **definedness of an extension** E **under** AE as AE/E, to specify that traversals of E in S' are defined. Formally, for all join states s of S', i.e. where an entry event is defined, and some input stream i, where the first element of i is an entry element, and either the last element is the first exit event in i, or no exit event occurs in i, we have: If $AE(i)$ holds, then here is a defined traversal (s, i, o) for some output sequence o. As the extension

is entry-exit triggered, the definition entails that E returns to a state in S for any exit event.

Assume S with input events I is extended to S' with some extension E with entry events E_{en} and exit events E_{ex}. For A/S and AE/E, we define the **extension assumptions** $EA(A, E)$ as follows: $EA(A, E)(i)$ holds if

(i) $el(i)$ is a stream over I and $A(el(i))$ holds and
(ii) for any occurrence of an entry event $en \in E_{en}$ in i of the form $i_0 :: en : e :: ex$, where $ex \in E_{ex}$ and e has no exit event, then S' is defined for $i_0 :: en$ and AE holds for $en : e :: ex$.

Intuitively, $EA(A, E)$ has to ensure the following. First, under elimination $EA(A, E)$ has to hold if A holds. Secondly, for the traversal of the new extension, the assumptions for AE have to hold. Note that the first condition, i.e. that $el(i)$ is a trace over I, ensures that new segments in the trace with events not in I are properly started with an entry event and terminated by an exit event. Otherwise, the elimination results in a trace which has events not in I.

We define E to be a **conservative extension** of S if it does not modify variables of S. In other words, the newly added transitions do not modify the variables in S. For conservative extensions, we can show that they do not modify behavior of the extended SD. It may still happen that the control flow does not return from the extension to the original SD.

The following theorem shows that an extended SD is defined for the traces in the extension assumptions $EA(A, E)$.

Theorem 1. *Assume S is extended to S' with some conservative, entry-exit triggered extension E. If A/S and AE/E, then $EA(A, E)/S'$.*

Proof. *The proof proceeds by induction on input streams. Assume i is an input of S'. The definedness of S' under $EA(A, E)$ follows from the cases as in the definition of $EA(A, E)$ as follows. In case the input only has elements from I, the case is trivial. In case an entry event occurs in i with S being at a state s, we have to show that S' is defined, which follows from the definition of $EA(A, E)$. Then the traversal in the extension is defined as AE holds for any state and any variable valuation. In case the traversal returns by some exit event to s, we observe that the traversal has not changed the variables of S as it is conservative. Hence the execution of S' at state s continues as S would in this state. As $A(el(i))$ holds, we can infer that also S' is defined until the next occurrence of an entry event. In case the traversal does not return, AE ensures definedness.* □

For a property A over streams we say A is **input-consistent**, if $A(i)$ implies $A(i')$ for all prefixes i' of i, i.e. there exists an i'' with $i = i' :: i''$. This assumption is needed for weak refinements, as non-termination may occur in an SD in an extension. Consider an input i which is permitted for the base SD. Then, in an extension an entry event may occur at i', which is a prefix of i. The elimination on the trace of the extended SD will cut off the trace after i' in case of divergence. For the refinement to hold, i' must then also be permitted in the assumptions for the base SD.

Theorem 2. *Assume S is extended to S' with some conservative and entry-exit triggered extension E and A is an input-consistent property. If A/S and AE/E, then $EA(A,E)/S'$ is a weak refinement of S.*

Proof. *The proof proceeds by induction on the input stream i of S' and follows the proof of the above theorem. In case E has no entry events, the case is trivial. We show $el(i, S'(i)) = (el(i), S(el(i)))$ inductively over the stream of entry events in i. Assuming it holds for a prefix of i_o of i and $i = i_0 : e_{entry} : i'$. As in the above proof, we relate the execution in S' with S under the elimination. In case a traversal of the extension returns, S' behaves as S as in the proof above after the return, and we can show the equation easily. In case of divergence, there is no exit event in i' and we have $el(i) = i_0$. Then also $el(i, S'(i)) = (el(i), S(el(i))) = (i_0, S(i_0))$. Furthermore, we have to show $A(el(i))$ follows from $A'(i)$. The critical case here is when a trace diverges in an extension, as el will cut off after the last entry event. Here, the assumption on input-consistency is needed to show that $A(el(i))$ holds.* □

This theorem shows that there exist assumptions, i.e. $EA(A,E)$, for which an extension (with conditions as above) is a weak elimination-based refinement. Based on this, we can transfer safety properties of an SD to an extended SD as discussed above.

Another issue is to compute $EA(A,E)$ efficiently from the definition of $EA(A,E)$ in practice. The main problem for this is to determine all the input sequences for which a join state can be reached. If this is possible in an SD, we may also compute an effective representation of $EA(A,E)$ by composing input sequences. We illustrate this by the following example.

Consider the lock extension in Figure 1, which adds a new locked state and permits any input in the lock state. Let *Set* be the base SD, *Lock* the extension and *SetLock* be the full SD with the extension as in Figure 1. Then we use regular expressions to define assumption predicates, where the set of input sequences defined by the regular expression defines when the predicate holds.

We define A_{Set} =*(set : off)** and A_{Lock} = *lock : (set,off,lock)* * *:: unlock*. For the extended SD with the lock, we define $A_{SetLock}$ = *(set : (off :set)** *:: (lock : (set,off,lock)* * *:: unlock)** *:: off)** . We use these sets to denote assumptions predicates which hold for all streams in the corresponding set.

In this example, $A_{SetLock}$ is the extension assumption $EA(A_{Set}, A_{Lock})$ for A_{Set}/Set and A_{Lock} based on the above definition. Applying the elimination to $A_{SetLock}$ yields A_{Set} and *set : (off :set)** specifies the inputs leading to the join points (here state B). Based on this, we have $A_{SetLock}/SetLock$ is a weak refinement of A_{Set}/Set.

Notice that we permit that a traversal remains infinitely long in the extension, which we consider as divergence when viewed from a refinement perspective of the base SD. Thus, for $A'_{Lock} = A_{Lock} \cup (lock^{\omega})$, where $lock^{\omega}$ denotes the infinite stream of *lock* events, we still have weak elimination-based refinement assuming $A'_{Lock}/Lock$ (instead of $A_{Lock}/Lock$).

If an extension terminates under specific assumptions, then we can show the stronger property of simulations. For an entry-exit triggered extension E, we say

AE/E terminates if there is not infinite traversal through E which is permitted by AE. For instance, in the lock example, there are possible traversals which stay infinitely long in the extension if no *unlock* event occurs. This can however be disallowed by the assumption AE.

Theorem 3. *Assume S is extended to S' with some conservative and entry-exit triggered extension E. If A/S and AE/E, and further AE/E terminates, then $EA(A,E)/S'$ is a strong refinement of S.*

The proof proceeds as the above result on weak refinement. Here, the proof is easier as all traversals permitted for the extension terminate.

Following the example above, we have $A_{SetLock}/SetLock$ is also a strong elimination-based refinement of A_{Set}/Set with extension $A_{Lock}/Lock$, as all permitted input sequences in A_{Lock} are finite and return to state B.

Recall that strong elimination-based refinement preserves liveness properties. Here, an example is the property that o occurs eventually. This holds, based on strong refinement for $A_{Lock}/Lock$. It does not hold assuming $A'_{Lock}/Lock$, as the traversals may remain infinitely long at the lock state.

5 Related Work

In the following, we discuss related work on statechart refinement and related concepts like UML state machines and other automata models.

Recently, the concept of eliminations was introduced with a focus on compatibility [13], in a setting of non-deterministic SDs with chaos semantics. The approach was also defining eliminations based on traversals of the feature extension, not purely on the trace level as done here. Based on an analysis of the traversals and SD internal states, it was shown when such extensions are a refinement in the sense presented here. Here, we are able to show results on definedness and strong refinement, where the assumptions assure the termination of the extension. This is not possible in prior approaches.

Earlier work on statechart refinement [16][7][15], which is using similar semantic models of statecharts, has developed several rules for refinement. The work in [14] develops a refinement calculus for statecharts as in [16] based on a mapping to the Z language. 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 [3]. Other work on UML in [19], which builds on concepts for object lifecycle modeling [17], considers the problem of consistent inheritance and observation consistency, which are similar to our notion of compatibility. In all of the above, refinement relations are defined by simply removing the new events or ignoring behavior after new events.

For related work on UML modeling, the concepts developed in [18] essentially cover basic cases of refining a state into several ones, which is different and not covered here. The work in [11] focuses on modeling the added features as independent and modular entities, modeled as statechart fragments.

Other work on modularity for model checking [2][10] 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. Similar goals have been pursed in the context of aspect-modeling for state machines, as shown in [20].

There is also recent work on compatibility for interface automata [1, 4]. In contrast to interface automata, we model the assumptions of an SD separately. This is conceptually similar to the work on interface input/output automata in [8], which also uses a separate model to describe the input assumptions. The assumptions are modeled as interface automata, which is just a more specific way to denote the input assumptions. However [8] does not focus on refinement and modular reasoning about definedness. More recent work on modal interface automata [9, 6] considers refinement more explicitly by modalities transitions which describe the possible, later refinements. This is different from our work, as we aim at adding behavior without requiring limitations on the SD to be extended.

Compared to our approach of using simply predicates, the work in [8] is using interfaces automata in a more specific way to denote the input assumptions. Unlike [8], we focus more on semantic refinement and modular reasoning about definedness.

6 Conclusions

In this paper, we have presented a new approach to reason about extensions of state transition diagrams based on assume/guarantee specifications. We have focused on extensions which only add new behavior, similar to observer aspects or conservative extensions on a programming language level. A particular problem is that new features may have additional input and output events, but may also reuse existing events. This makes it difficult to reason for what input an extended SD is defined and when it preserves the original behavior. Due to this, existing notions of refinement do not apply. Here, we have developed a new approach towards refinement which allows one to reason about such extended state transition diagrams in a modular way.

In particular, we have developed new refinement concepts for weak and strong refinements, based on an elimination of the newly added behavior on the trace level. These eliminations can be seen as a generalization of typical abstractions, which only remove new input/output events of an extension. Secondly, we have presented an approach for compositional reasoning of such extended SDs using a assume-guarantee calculus. Based on assumptions for the base SD and the extension, we can show when an SD is defined and property preserving after adding the extension. In detail, we show when adding a new feature adds only additional behavior, possibly with divergence. Similar to the considerable work on

property preserving aspects and features on the programming language level, we have captured typical extensions like observers also for state transition diagrams in our new approach.

Our approach based on assumptions and guarantees can express various properties of such SDs. We have illustrated that our results can be used to preserve safety and liveness properties when extending an SD. Further work is needed to study in detail how to model and validate typical safety of liveness properties in this form. Also, further work will address how to compute the assumptions needed for an extended SD in an effective way.

Acknowledgements. The author would like to thank Peter Scholz, Martin Wirsing, Sebastian Bauer and Rolf Hennicker for discussion and feedback on this work.

References

[1] Alfaro, L., Henzinger, T.: Interface-based design. In: Broy, M., Grünbauer, J., Harel, D., Hoare, T. (eds.) Engineering Theories of Software Intensive Systems. NATO Science Series, vol. 195, pp. 83–104. Springer, Netherlands (2005)

[2] Blundell, C., Fisler, K., Krishnamurthi, S., Van Hentenrvck, P.: Parameterized interfaces for open system verification of product lines. In: Proceedings of the 19th International Conference on Automated Software Engineering, pp. 258–267 (September 2004)

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

[4] David, A., Larsen, K.G., Legay, A., Nyman, U., Wasowski, A.: 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 2010, pp. 91–100. ACM, New York (2010)

[5] Djoko, S.D., Douence, R., Fradet, P.: Aspects preserving properties. In: Proceedings of the 2008 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation, PEPM 2008, pp. 135–145. ACM, New York (2008)

[6] Fischbein, D., Uchitel, S., Braberman, V.: 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 2006, pp. 39–48. ACM, New York (2006)

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

[8] Larsen, K.G., Nyman, U., Wąsowski, A.: Interface input/output automata. In: Misra, J., Nipkow, T., Sekerinski, E. (eds.) FM 2006. LNCS, vol. 4085, pp. 82–97. Springer, Heidelberg (2006)

[9] Larsen, K.G., Nyman, U., Wąsowski, A.: Modal I/O automata for interface and product line theories. In: De Nicola, R. (ed.) ESOP 2007. LNCS, vol. 4421, pp. 64–79. Springer, Heidelberg (2007)

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

[11] Prehofer, C.: Plug-and-play composition of features and feature interactions with statechart diagrams. Software and Systems Modeling 3, 221–234 (2004)
[12] Prehofer, C.: Semantic reasoning about feature composition via multiple aspect-weavings. In: Proceedings of the 5th International Conference on Generative Programming and Component Engineering, GPCE 2006, pp. 237–242. ACM, New York (2006)
[13] Prehofer, C.: Behavioral refinement and compatibility of statechart extensions. In: Workshop on Formal Engineering approaches to Software Components and Architectures. I Electronic Notes in Theoretical Computer Science (ENTCS) (2012)
[14] Reeve, G., Reeves, S.: Logic and refinement for charts. In: Proceedings of the 29th Australasian Computer Science Conference, ACSC 2006, vol. 48, pp. 13–23. Australian Computer Society, Inc., Darlinghurst (2006)
[15] Rumpe, B., Klein, C.: Automata describing object behavior. In: Specification of Behavioral Semantics in Object-Oriented Information Modeling, pp. 265–286. Kluwer Academic Publishers (1996)
[16] Scholz, P.: Incremental design of statechart specifications. Science of Computer Programming 40(1), 119–145 (2001)
[17] Schrefl, M., Stumptner, M.: Behavior-consistent specialization of object life cycles. ACM Trans. Softw. Eng. Methodol. 11, 92–148 (2002)
[18] Simons, A.J.H., Stannett, M.P., Bogdanov, K.E., Holcombe, W.M.L.: W.M.l.: Plug and play safely: Rules for behavioural compatibility. In: IProc. 6th IASTED Int. Conf. Software Engineering and Applications, pp. 263–268 (2002)
[19] Stumptner, M., Schrefl, M.: Behavior consistent inheritance in UML. In: Laender, A.H.F., Liddle, S.W., Storey, V.C. (eds.) ER 2000. LNCS, vol. 1920, pp. 527–542. Springer, Heidelberg (2000)
[20] Zhang, G., Hölzl, M.: Hila: High-level aspects for uml state machines. In: Ghosh, S. (ed.) MODELS 2009. LNCS, vol. 6002, pp. 104–118. Springer, Heidelberg (2010)